Dr. Sebastian Küpper

Grundlagen der Informatik 2

Modul 65002

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Einführung in die Informatik 2

Sebastian Küpper

26. Februar 2025

Inhaltsverzeichnis

Lektion 1: Objektorientierung, Algorithmik und Theoretische Grenzen

Das vorliegende Kursmaterial stellt den zweiten Teil der Einführung in die Informatik für Studenten der Wirtschaftsinformatik dar. Im ersten Kurs haben Sie bereits gelernt, einfache Probleme durch Programmierung zu lösen, den Aufbau eines Computers und die Funktionsweise von Rechnernetzen kennen gelernt. Darüber hinaus haben Sie die Rolle des Betriebssystems kennengelernt. Ziel dieses Kurses ist es, Ihre Kenntnisse in der Programmierung zu vertiefen und theoretisch einzuordnen. In dieser ersten Lektion werden wir uns darauf konzentrieren, die konzeptuellen Grundlagen für die übrigen sechs Lektionen zu legen, um dem Umstand Rechnung zu tragen, dass Beleger dieses Kurses sehr unterschiedliche Voraussetzungen hinsichtlich der mathematisch-informatischen Vorbildung mitbringen.

Die weiteren sechs Lektionen befassen sich mit drei unterschiedlichen, aber doch verknüpften Themenfeldern. In den Lektionen 2 und 3 besprechen wir das Konzept der objektorientierten Programmierung und dessen Umsetzung in C#. Um einen Ausblick auf unterschiedliche Lösungsansätze in unterschiedlichen Programmiersprachen zu geben, ziehen wir immer dann, wenn sich die Umsetzung in C# von der in Java unterscheidet, einen entsprechenden Vergleich. Wenngleich C# als Leitbeispiel eine Programmiersprache ist, in der Sie sich am Ende des Kurses so gut auskennen sollten, dass Sie komplexe Probleme in der Sprache lösen können, versteht sich dieser Kurs nicht als C#-Programmierkurs. Stattdessen sollen die grundlegenden Konzepte verstanden werden, so dass Sie im Arbeitsalltag in die Lage versetzt werden, sich auf eine Vielzahl von Programmiersprachen mit verwandten Konzepten weitgehend reibungslos einzustellen.

Bei der objektorientierten Programmierung handelt es sich um ein Modellierungskonzept, das dabei hilft, Daten und zugehörige Operationen zu organisieren, Informationen zu kapseln und eine Abstraktionsebene in der Interaktion zwischen Programmteilen zu schaffen. Auf diese Weise kann, sofern man die verschiedenen Werkzeuge der Objektorientierung sinnvoll nutzt, die gedankliche Komplexität deutlich reduziert werden. Die Kernidee hinter der Objektorientierung ist es, den Untersuchungsgegenstand hinsichtlich gleichartigen Akteuren zu analysieren. Diese gleichartigen Akteure werden zu Klassen zusammengefasst, die einen prototypischen Bauplan für Akteure dieses Types darstellen. Die einzelnen Akteure nennen sich Objekte und besitzen vermöge ihrer Klasse einen Objektzustand und Operationen, die auf diesem Objektzustand vorgenommen werden können. Das Geheimnisprinzip stellt sicher, dass nur ein Objekt selbst unmittelbar seinen Objektzustand beeinflussen kann und Änderungen am Objekt nur über Nachrichten an das Objekt mit-

tels der angebotenen Operationen vorgenommen werden können. Ein objektorientiertes Programm organisiert also die Kommunikation zwischen individuellen Akteuren, die Rollen oder Akteure im Anwendungskontext repräsentieren. Um diesen Ansatz zu unterstützen, wurden verschiedene Konzepte wie Vererbung, Polymorphie und Sichtbarkeitsklassen entwickelt, die wir in den Lektionen 2 und 3 ausführlich beleuchten wollen.

Während die Objektorientierung einen Rahmen dafür bietet, wie Daten organisiert werden können und wie die Software aus einer abstrakten Perspektive strukturiert werden kann, bietet sie für Probleme, die über die reine Datenverwaltung hinaus gehen, keine unmittelbaren Lösungen. Wenn Daten nicht nur erhoben und verwaltet werden sollen, sondern auch umfassend bearbeitet, muss man die abstrakte Perspektive verlassen. In den Lektionen 4 und 5 konzentrieren wir uns auf Algorithmen und Datenstrukturen, die es erlauben, komplexe Berechnungen effizient durchzuführen. Wenngleich die algorithmische Perspektive kein Augenmerk auf objektorientiere Modellierung legt, sollte das nicht als konträrer Ansatz zur objektorientierten Programmierung verstanden werden, sondern als komplementärer. Während die Objektorientierung sich auf die Struktur des Programms konzentriert, gibt die Algorithmik Ihnen die Techniken an die Hand, um die Operationen mit Leben zu füllen und sie sowohl effektiv, das heißt zielorientiert, als auch effizient, das heißt ressourcenschonend, umzusetzen.

Ein besonderes Augenmerk legen wir bei unseren algorithmischen Überlegungen auf das Konzept der Rekursion, das es erlaubt, kompakte und elegante Lösungen für komplexe Probleme zu formulieren. Die dynamische Programmierung wird als Optimierungstechnik für rekursive Algorithmen erläutert. Die Kernidee ist hier, im Tausch gegen zusätzlichen Speicherplatz eine – idealerweise um Klassen – kürzere Laufzeit zu erzielen. Das klassische Anwendungsfeld dynamischer Programmierung ist die Suche nach einem Optimum, das heißt einer größten oder kleinsten Lösung für ein Problem – eine Fragestellung, die sich gerade in der Wirtschaftsinformatik qua ihres Anwendungsfeldes immer wieder finden wird. Ein wichtiges Hilfsmittel für die Algorithmik stellen effiziente und problemangepasste Datenstrukturen dar. Ein besonderer Schwerpunkt wird in dieser Lektion auf Graph-basierte Datenstrukturen wie den binären Suchbaum gelegt.

Hat man sich ausführlich mit algorithmischen Techniken befasst, könnte man der Idee erliegen, dass sich für jedes denkbare Problem, genügend Gedankenarbeit vorausgesetzt, eine Lösung oder sogar eine effiziente Lösung finden lassen kann. Dass dies ein Irrglaube ist und sogar vermeintlich einfache Fragestellungen wie „Wird dieses Programm jemals eine 1 ausgeben?“ nicht algorithmisch beantwortet werden können, werden wir in dem Themenkomplex Theoretische Informatik in den Lektionen 6 und 7 erfahren. Hier wird der Beweis erbracht, dass es Probleme gibt, die sich grundsätzlich der algorithmischen Lösung in ihrer Allgemeinheit entziehen und dann eine Beweistechnik erarbeitet, mit der man deutlich einfacher für weitere Probleme nachweisen kann, dass sie nicht entscheidbar, das heißt nicht algorithmisch lösbar, sind. Doch Lösbarkeit allein ist in vielen Fällen nicht ausreichend. Eine effektive Lösung zu finden ist gut, aber wenn für interessante Problemgrößen die Berechnungen länger dauern als die eigene Lebenserwartung, hält sich die praktische Nutzbarkeit in Grenzen. Auf der vermeintlichen Schwelle zwischen effizient und nicht effizient lösbaren Problemen stehen die sogenannten NP-vollständigen Probleme. Ebenfalls in Lektion 6 werden wir eine kurze Einführung in die Komplexitätstheorie geben, das ist

das Feld der Informatik, das sich damit befasst, die inhärente Schwierigkeit (grundsätzlich lösbarer) Probleme einzugrenzen und zu beschreiben.

Nachdem unsere hoffnungsvolle Sicht auf die Algorithmik hinlänglich getrübt wurde, stellt sich die Frage, wie wir mit den Grenzen der Berechenbarkeit und Komplexität umgehen sollen. Hierzu betrachten wir schließlich in der siebten und letzten Lektion Modellierungstechniken und Sprachklassen, die es ermöglichen, Probleme sinnvoll einzuschränken. In vielen Fällen müssen wir unentscheidbare Fragestellungen nämlich gar nicht in aller Allgemeinheit lösen, sondern sind nur an bestimmten Probleminstanzen überhaupt interessiert. Kommen wir beispielsweise auf die Frage zurück, ob ein Programm jemals eine 1 ausgibt. Im Allgemeinen ist dieses Problem zwar unentscheidbar, aber wie ist es, wenn wir mehr über das Programm wissen? Beispielsweise, dass es ohne das Zwischenspeichern von Informationen auskommt? Oder dass es mit einem einzelnen Stapel auskommt? Mit solchen Einschränkungen werden viele Probleme, die in der allgemeinen Form unentscheidbar sind, nicht nur entscheidbar, sondern sogar effizient mit einfachen Algorithmen beherrschbar. Mit zwei besonders wichtigen Klassen von eingeschränkten Problemklassen, die regulären und die kontextfreien Sprachen, befassen wir uns in Lektion 7.

Selbstverständlich können die angesprochenen Themen nicht in ihrer vollen Komplexität besprochen werden. Es gibt spezialisierte Kurse an der Fakultät, die die einzelnen Themenbereiche noch deutlich umfassender beleuchten. Ich hoffe jedoch, Ihnen mit diesem Kurs ein solides, programmierorientiertes Grundverständnis für diese drei Felder zu vermitteln. Basierend auf Ihren Interessen können Sie dann, insbesondere im Master-Studium, vertiefende Kurse in den einzelnen Feldern belegen.

Eine kurze Einordnung zur persönlichen Anrede in diesem Kurs. Dieser Kurs ist im generischen Neutrum verfasst. Das bedeutet, dass im Text bei Ansprachen, die in der deutschen Sprache ein grammatikalisches Geschlecht besitzen, die Grundform, aber nicht mit dem standardsprachlich üblichen Maskulin, sondern dem Neutrum verwendet wird. Eine Beispielformulierung wäre „das Leser des Kurstextes“. Ich wähle diese Form, um alle Geschlechter gleich zu behandeln und gleichzeitig die Geschlechtlichkeit in den vielen Kontexten, in denen sie sachlich keine Rolle spielt, bewusst nicht zu betonen.

Schließlich noch ein Wort zu Quellenangaben. Da dieser Kurstext der Lehre dient und nicht der Darstellung aktueller wissenschaftlicher Erkenntnisse, wird es im laufenden Text keine Quellenangaben geben. Keine der hier dargestellten Erkenntnisse sind meine eigenen, sondern es handelt sich um eine Zusammenfassung wesentlicher Ideen der Informatik. Die wichtigsten Quellen für den Kurs seien an dieser Stelle aber einmal erwähnt.

- „Einführung in die Objektorientierte Programmierung“ von Bernd Krämer für die FernUniversität in Hagen in den Fassungen von 2014 und der letzten Überarbeitung durch mich aus dem Jahr 2022.

  Der vorliegende Kurs ist aus einer Neugestaltung des Kurses von Bernd Krämer hervorgegangen und enthält in den Lektionen 1 bis 5, schwerpunktmäßig in den Lektionen 2 und 3, viele Darstellungen und – mit freundlicher Genehmigung des Originalautors – auch nicht im Einzelnen gekennzeichnete wörtliche Übernahmen aus diesem Originaltext. Hinsichtlich der Einführung in die objektorientierte Programmierung kann der vorliegende Kurstext als Adaption, Anpassung und Modernisierung

des von Bernd Krämer verfassten Originaltextes verstanden werden.

- „C# documentation“ von Microsoft, zu finden zum Zeitpunkt des Schreibens dieses Skriptes unter der URL `https://learn.microsoft.com/en-us/dotnet/csharp/`.

  Die Dokumentation wurde immer dann, wenn C#-Sprachkonstrukte vorgestellt werden, zu Rate gezogen. Die Darstellung der Syntax folgt also im Wesentlichen dieser Dokumentation.

- „Algorithmen – Eine Einführung“ von Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein, 3. Auflage 2010, ISBN: 3486590022.

  Dieses Standardwerk der Algorithmik wurde – neben den existierenden Texten Bernd Krämers – wesentlich als Quelle für die Lektionen 4 und 5 herangezogen. Insbesondere die Darstellungen zur Dynamischen Programmierung orientieren sich an dem Buch. Allen Lesern, die mehr zu Algorithmik lernen möchten, sei dieses umfassende und didaktisch durchdachte Buch sehr ans Herz gelegt. Neben der hier verwendeten dritten deutschen Auflage sind auch frühere und spätere Auflagen, sowohl in Deutsch als auch Englisch, empfehlenswert.

- „Theoretische Informatik kurzgefasst“ von Uwe Schöning, 5. Auflage 2008, ISBN: 3827418240.

  Dieses Buch ist ein kompaktes aber gründliches Standardwerk zu den grundlegenden Themen der Theoretischen Informatik und ist eine der zwei Hauptquellen zu den Lektionen 6 und 7. Die Darstellungen weichen zwar deutlich von den hier gewählten Darstellungen ab, da dieser Kurs versucht, die theoretischen Inhalte im Anwendungskontext zu erläutern, die grundlegenden Beweisideen sind aber in aller Regel die gleichen.

- Skripte „Automaten und formale Sprachen“ und „Berechenbarkeit und Komplexität“ von Barbara König, Universität Duisburg-Essen, in den Fassungen von 2017.

  Einzelne Darstellungen in den Lektionen 6 und 7 folgen den Argumentationen in diesen beiden Skripten. Bei Beispielen orientiere ich mich – in Einzelfällen auch nahezu wörtlich und mit freundlicher Genehmigung der Originalautorin – an den Aufgabenstellungen dieser Kurse und zugehörigen Übungsangeboten, an denen ich in meinen fünf Jahren als Mitarbeiter am Lehrgebiet mitgewirkt habe.

1.1 Objektorientierung als Designkonzept

Objektorientierte Programmiersprachen sowie objektorientierte Erweiterungen von zuvor nicht objektorientierten Programmiersprachen werden von vielen Forschern auf dem Feld der Softwaretechnik und Programmierern in der Praxis bevorzugt eingesetzt. Ein wichtiger Grund hierfür ist, dass die Struktur der Problemdomäne strukturgebend für die zu entwickelnde Software verwendet werden kann. Ein genaues Verständnis der Domäne ergibt

robuste Modelle, die dann auch die Kommunikation zwischen Auftraggeber – in vielen Fällen selbst nicht oder nicht umfassend informatisch gebildet – und Entwickler vereinfachen können.

In dieser Lektion werden wir die grundlegenden Ideen der objektorientierten Programmierung zunächst auf einem abstrakten Level kennen lernen und mithilfe der Modellierungssprache UML ausdrücken. Wir werden hier noch auf eine programmtechnische Umsetzung verzichten und die Konzepte in den folgenden beiden Lektionen noch einmal ausführlich orientiert an C# besprechen.

Als Fallstudie werden wir die Modellierung einer Verwaltungssoftware für einen Blumenladen vornehmen, die dann auch als motivierendes Szenario durch die Lektionen 1 und 2 führt.

1.1.1 Objekte und Klassen

Der zentrale Begriff der objektorientierten Programmierung ist natürlich das Objekt:

Definition 1.1.1 : Objekt

Die objektorientierte Programmierung bezeichnet die Abbilder konkreter individuell unterscheidbarer Gegenstände und Träger individueller Rollen in Entwurfsdokumenten und Programmen als Objekte.

Um uns den Begriff des Objektes begreiflich zu machen, betrachten wir folgende Beschreibung einer Software zur Verwaltung eines Blumenladens.

Beispiel 1.1.2 : Blumenladen

Erstellt werden soll eine Verwaltungssoftware für einen Blumenladen, die die Verwaltung des Warenbestandes, inklusive der Abbildung von Kauf- und Verkaufsvorgängen erlaubt. Der Blumenladen bietet verschiedene Artikel, beispielsweise Schnittblumen, Topfpflanzen und Zubehör für Blumenfreunde an. Jeder Artikel verfügt über einen Preis, eine Artikelnummer zur eindeutigen Identifikation und für Bestellprozesse, sowie eine textuelle Beschreibung des Produkts. Manche Produkte wie die Schnittblumen verfügen zudem über ein Haltbarkeitsdatum. Um Bestellungen entgegenzunehmen und an die richtigen Kunden zu liefern, werden derzeit in einer Kundenkartei Name und Adresse der Kunden hinterlegt.

Für jeden Kaufvorgang muss eine nummerierte Rechnung erstellt werden, die neben dem Rechnungsempfänger den Nettopreis, die anfallende Mehrwertsteuer und einen möglicherweise verrechneten Rabatt, zum Beispiel weil ein Kunde regelmäßiger Kunde ist oder eine besonders umfangreiche Bestllung vornimmt, erfassen muss.

In diesem Szenario wollen wir nun beispielhaft einige Objekte erfassen:

Beispiel 1.1.3 : Objekte im Blumenladen

Einige Beispiele für Objekte im Blumenladen sind:

- Ein Kunde mit Namen Alan Lovelace
- Eine Kundin mit dem Namen Ada Turing
- Eine Gerbera-Schnittblume, die die Artikelnummer 3 hat und 14 Cent kostet
- Eine Rechnung für Ada Turing über 159€ mit der Rechnungsnummer 2653

Es ist anzumerken, dass Objekte nur konkrete Individuen bezeichnen. Eine allgemeine Schablone (zum Beispiel „Rechnung“) ist kein Objekt.

Bei der Modellbildung führen wir eine Abstraktion der Objekte in der Domäne durch. So haben die einzelnen Kunden im Blumenladen zahlreiche Eigenschaften, beispielsweise eine Körperhöhe, eine Körpermasse oder die Fähigkeit, zu springen, die im Kontext unseres Modells keine Rolle spielen. Wir bilden nur diejenigen Eigenschaften und Fähigkeiten ab, die im Anwendungskontext relevant sind.

Allerdings wäre es kaum praktikalbel, händisch jedes Objekt zu programmieren. Man stelle sich vor, jedes Mal, wenn ein neues Kunde in die Kundendatenbank aufgenommen werden soll, müsste ein Programmierer die Repräsentation des Kunden programmieren. Stattdessen betrachten wir eine Abstraktion gleichartiger Objekte, die Klasse.

Definition 1.1.4 : Klasse

Eine Klasse beschreibt in der objektorientierten Programmierung einen Bauplan für viele ähnliche, aber individuell unterscheidbare Objekte.

Klassen werden durch Substantive bezeichnet.

Bemerkung 1.1.5

Jedes Objekt ist eine Ausprägung einer bestimmtem Klasse. Objekte werden gelegentlich auch als Exemplare oder Instanzen bezeichnet.

Beispiel 1.1.6 : Klassen

Beispiele für Klassen in unserem Beispiel wären „*Kunde*“ und „*Rechnung*“. Die Kunden Alan Lovelace und Ada Turing sind dann Instanzen der *Klasse Kunde* und die Rechnung mit der Nummer 2653 eine Instanz der Klasse *Rechnung*.

Wenn wir Klassen programmieren, werden wir in den nachfolgenden Lektionen

englische Namen beginnend mit Großbuchstaben für Klassen verwenden, damit deren Benennung sprachlich zu den vorgefertigen Klassen in C# und Java passen.

Klassen beschreiben die Eigenschaften (Attribute), die alle Instanzen dieser Klasse besitzen, und auch deren Verhalten und Fähigkeiten (Methoden).

Definition 1.1.7 : Attribut

Die Eigenschaften, die alle Objekte einer Klasse besitzen, werden durch Attribute dargestellt.

Attribute werden durch Substantive bezeichnet und wir schreiben die Attribute stets mit einem kleinen Anfangsbuchstaben.

Beispiel 1.1.8

Attribute der Klasse *Kunde* sind *name* und *adresse*. Eine *Rechnung* besitzt beispielsweise die Attribute *rechnungsnummer*, *betrag* und *rabatt*.

Attribute beschreiben lediglich, welche Eigenschaften ein Objekt (Exemplar) einer Klasse hat, jedoch nicht welchen konkreten Wert diese Eigenschaft bei dem gegebenen Objekt besitzt. Die konkreten Ausprägungen, die die Attribute in einem Objekt einnehmen, bezeichnen wir als Attributwert:

Definition 1.1.9 : Attributwert

Ein Attributwert bezeichnet den Wert, den ein Objekt für ein bestimmes Attribut besitzt.

Die Gesamtheit der Attributwerte eines Objektes nennen wir den Objektzustand.

Beispiel 1.1.10

Ein Beispiel für den Attributwert *name* der Klasse *Kunde* ist „Ada Turing“ und ein Beispiel für einen Attributwert des Attributes *rechnungsnummer* der Klasse *Rechnung* ist 2653.

Die Methoden einer Klasse erlauben die Manipulation eines Objektes:

Definition 1.1.11 : Methode

Das gemeinsame Verhalten aller Objekte einer Klasse wird durch die Methoden der Klasse bestimmt. Jede Methode beschreibt eine Fähigkeit oder mögliche Form des Umgangs mit einem Objekt der Klasse. Methoden werden durch Verben bezeichnet. Abweichend von der Konvention in C# (aber der Konvention in Java, C++ und Delphi

folgend) beginnen wir Methoden mit einem kleinen Buchstaben. In C# ist es üblich, die Notation einer Methode mit einem Großbuchstaben zu beginnen.

Beispiel 1.1.12

Ein Beispiel für eine Methode der Klasse *Rechnung* ist *setzeMWSt*.

1.1.2 Beziehungen

Objekte können Beziehungen zu Objekten der gleichen oder anderer Klassen haben.

Definition 1.1.13 : Assoziation und Verknüpfung

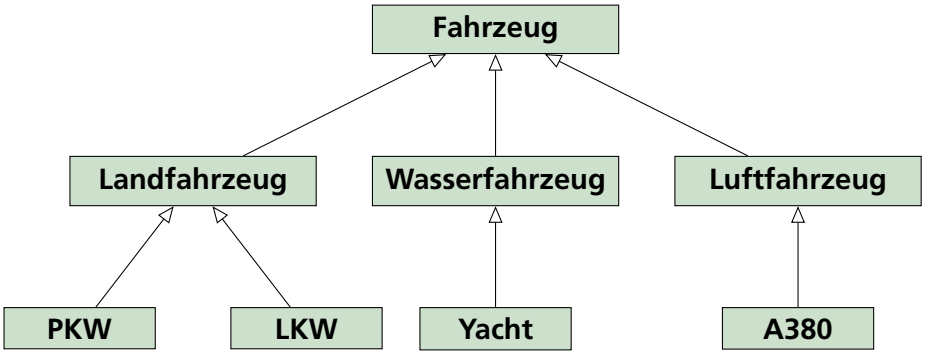
Eine Assoziation zwischen zwei Klassen drückt aus, dass es zwischen Objekten dieser Klassen eine Beziehung geben kann. Eine Assoziation besitzt einen Namen. Die konkrete Beziehung zwischen zwei Instanzen wird Verknüpfung genannt. Zu jeder Verknüpfung zwischen zwei Instanzen muss es immer eine zugehörige Assoziation zwischen den beiden Klassen geben.

Beispiel 1.1.14

Jede *Rechnung* wird für einen *Kunden* ausgestellt. Somit hat ein Objekt der Klasse *Rechnung* eine Verknüpfung zu einem Objekt der Klasse *Kunde*. Zwischen den beiden Klassen *Kunde* und *Rechnung* existiert die Assoziation *rechnungsempfaenger*. Die Rechnung 2653 kann somit eine Verknüpfung zum Kunden Alan Lovelace besitzen.

Eine wichtige Relation zwischen Klassen ist die Spezialisierung. Wir haben bereits die Klasse *Artikel* kennengelernt, die verschiedene Ausprägungen, beispielsweise Schnittblumen und Zubehör, haben kann. Zwar könnte man je nach Modellierungsanspruch darauf verzichten, Schnittblumen und Zubehör programmatisch zu unterscheiden, in vielen Fällen geht die Spezialisierung aber damit einher, dass zusätzliche wichtige Eigenschaften zu beachten sind – im Fall der Schnittblume beispielsweise das Verfallsdatum. Insofern ist das Konzept der Generalisierung und Spezialisierung bei der objektorientierten Programmierung ein wesentliches.

In der Realität kennen wir viele solche Beziehungen. So können wir beispielsweise allgemein von Fahrzeugen sprechen. Allerdings ist es in machen Situtationen notwendig, zwischen PKWs und LKWs zu unterscheiden. PKWs und LKWs sind also spezielle Fahrzeuge. Fahrzeuge können außerdem in Land- und Wasserfahrzeuge unterschieden werden. Wenn wir das Modell noch weiter ergänzen, erhalten wir eine ganze Hierarchie:



Solche Hierarchien treffen wir auch häufig in der Biologie an, wenn Tiere oder Pflanzen in einer bestimmten Systematik erfasst werden.

Durch solche Spezialisierungen drücken wir aus, dass Klassen gemeinsame Eigenschaften und Verhalten aufweisen. Wir sprechen dann davon, dass die speziellere Klasse die Eigenschaften und das Verhalten der allgemeineren erbt. Dabei kann die speziellere Klasse immer noch neue Eigenschaften und Verhalten hinzufügen. Manchmal ist es nötig, dass das Verhalten der spezielleren Klassen angepasst wird. Wenn das Verhalten verändert wird, dann sprechen wir davon, dass das Verhalten bzw. die Methode überschrieben wird. Eine Klasse B kann nur dann eine Spezialisierung der Klasse A sein, wenn gilt, dass jedes Objekt der Klasse B ein A ist.

Beispiel 1.1.15

Jeder PKW ist ein Fahrzeug, umgekehrt gilt dies jedoch nicht. Genauso wenig gilt die Aussage, dass jeder PKW ein LKW ist.

Definition 1.1.16 : Gernalisierung

Eine Generalisierung zwischen zwei Klassen drückt aus, dass die speziellere Klasse die Eigenschaften, das Verhalten und die Beziehungen der Allgemeineren erbt und dabei erweitern oder überschreiben kann. Ein Objekt der spezielleren Klasse kann überall dort verwendet werden, wo ein Objekt der allgemeineren Klasse verwendet werden kann.

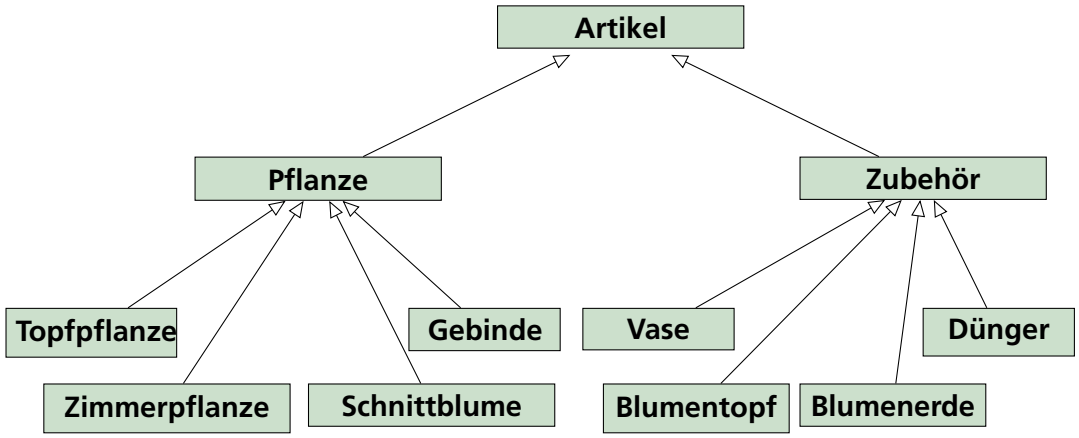
Bemerkung 1.1.17

Die allgemeinere Klasse wird in der Regel als Oberklasse oder Elterklasse (in älteren Texten oft auch Vaterklasse) und die speziellere als Unterklasse oder Kindklasse bezeichnet. Der Umstand, dass die Unterklasse von der Oberklasse Eigenschaften und Verhalten übernimmt, wird auch Vererbung genannt.

Selbsttestaufgabe 1.1.1

Versuchen Sie, aus der folgenden Beschreibung eine Klassenhierarchie für die im Blumenladen verkauften Artikel zu erstellen: Der Blumenladen verkauft sowohl Pflanzen als auch Zubehör. Beim Zubehör gibt es unter anderem Blumentöpfe, Vasen, Blumenerde und Dünger. Bei den Pflanzen wird zwischen Topfpflanzen, Zimmerpflanzen und Schnittblumen unterschieden. Zudem werden auch Sträuße verkauft.
(Attribute und Methoden müssen Sie nicht berücksichtigen.)

Musterlösung zu Selbsttestaufgabe 1.1.1



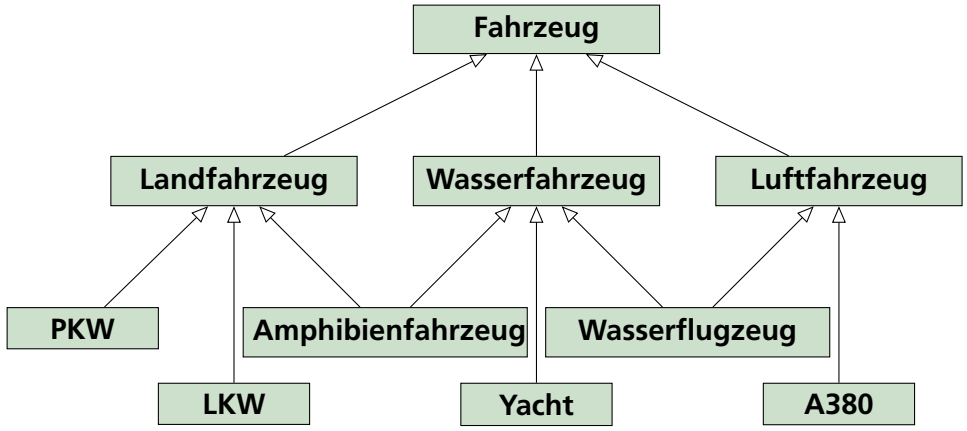
In der realen Welt kommt es vor, dass eine Klasse die Spezialisierung mehrerer Klassen ist. So ist ein Amphibienfahrzeug beispielsweise sowohl ein Land- als auch ein Wasserfahrzeug. Die Klasse Amphibienfahrzeug ist somit eine Spezialisierung der Klasse Land- und der Klasse Wasserfahrzeug. In einem solchen Fall spricht man von Mehrfachvererbung.

Definition 1.1.18 : Mehrfachvererbung

Wenn eine Klasse die Spezialisierung mehrerer anderer Klassen ist, so nennen wir dies Mehrfachvererbung.

Beispiel 1.1.19

Wir haben unsere Fahrzeughierarchie um zwei Klassen erweitert, bei denen Mehrfachvererbung geboten erscheint.



Selbsttestaufgabe 1.1.2

Ordnen Sie die folgenden Begriffe den verschiedenen Konzepten, wie Klasse, Objekt, Attribut, Attributwert und Methode, zu:

Rose, Rechnung, passeBeschreibungAn, Artikel, Preis, Rechnungsnummer, Farbe, ueberprüfeLagerBestand, Hanna Meier, "Zur Ecke 17, 12345 Irgendwo", Kunde, Rot, Name, Schnittblume, 31415, ändereName, Adresse

Musterlösung zu Selbsttestaufgabe 1.1.2

Rose (Klasse), Rechnung (Klasse), passeBeschreibungAn (Methode), Artikel (Klasse), Preis (Attribut), Rechnungsnummer (Attribut), Farbe (Attribut), ueberprüfeLagerBestand (Methode), Hanna Meier (Objekt), "Zur Ecke 17, 12345 Irgendwo" (Attributwert), Kunde (Klasse), Rot (Attributwert), Name (Attribut), Schnittblume (Klasse), 31415 (Attributwert), ändereName (Methode), Adresse (Attribut)

1.1.3 Kommunikation

In einem objektorientierten Programm existieren viele verschiedene Objekte. Damit das Programm das gewünschte Ergebnis erreichen kann, ist es nötig, dass die Objekte zusammenarbeiten.

Die Zusammenarbeit von Objekten läuft mit Hilfe von Nachrichten ab. Ein Objekt kann eine Nachricht an ein anderes Objekt oder gelegentlich auch sich selbst verschicken. Eine solche Nachricht enthält die Information, welche Methode ausgeführt werden soll, und wenn nötig noch weitere Informationen, die das empfangende Objekt benötigt, um die Nachricht zu verarbeiten. Das versendende Objekt wartet in der sequenziellen Programmierung so lange, bis das empfangende Objekt eine Antwort sendet, und setzt erst dann seine Arbeit fort.

Beispiel 1.1.20

Stellen wir uns vor, ein Kunde will einen Strauß Blumen kaufen. Durch die Eingabe des Verkäufers wird das für einen Verkauf zuständige Objekt zunächst eine Nachricht an das Lagerobjekt verschicken. Das Lagerobjekt wird aufgefordert zu überprüfen, ob alle benötigten Blumen in ausreichender Anzahl verfügbar sind, und, wenn dies der Fall ist, die benötigten Blumen aus dem Lager zu entnehmen. Anschließend wird ein neues Rechnungsobjekt erzeugt. Dieses erhält die Nachricht, den Blumenstrauß als Rechnungsposten hinzuzufügen. Sodann kann das Rechnungsobjekt nach dem endgültigen Preis gefragt werden und nach erfolgreicher Bezahlung darüber informiert werden.

Bemerkung 1.1.21

In diesem Kurs werden wir uns nur mit sequenzieller Programmierung beschäftigen. In der sequenziellen Programmierung wird zu jedem Zeitpunkt des Programms nur eine Aktion ausgeführt und deren Reihenfolge ist fest vorgegeben. In der nebenläufigen oder parallelen Programmierung können hingegen mehrere Aktionen gleichzeitig oder in nicht vorhersehbarer Reihenfolge ausgeführt werden.

1.1.4 Objektorientierte Analyse und Entwurf

Wir haben gelernt, dass wir Gegenstände, die wir als gleichartig ansehen, zu Klassen und somit zu einem Begriff zusammenfassen können. Die Klassenbildung ist meist nicht eindeutig. Sie hängt vom Verständnis der beteiligten Personen ab.

Jede Beschreibung der Realität hängt von der Wahl der Fachbegriffe und einer vorsichtigen Erläuterung der Phänomene ab, die jeder Begriff bezeichnet. Wichtig ist, dass wir uns vor dem Entwurf eines Programmsystems ein genaues Bild der Wirklichkeit verschaffen und dieses auch mit viel Disziplin umfassend und möglichst eindeutig beschreiben.

Unsere Wahrnehmung der Wirklichkeit müssen wir nachvollziehbar mit den schematischen und formalen Beschreibungen, die ein Programm ausmachen, in Beziehung setzen. Nur so können wir sicher sein, dass die Auswirkungen des Programms die gewünschten Resultate liefern. In jeder Anwendungswelt werden Begriffe benutzt, die von vorne herein nur selten von Entwicklern voll verstanden werden.

Beispiel 1.1.22

Ein Unfall einer Lufthansa-Maschine beim Landeanflug konnte eindeutig auf die unvollständige Modellierung der Phänomene der Wirklichkeit zurückgeführt werden:

Das Bremssystem dieser Maschine war so ausgelegt, dass der Umkehrschub, der ein Flugzeug nach der Landung stark abbremst, erst eingeschaltet werden kann, wenn die Maschine tatsächlich gelandet ist. Dadurch sollte ein versehentliches Einschalten

des Umkehrschubs in der Luft verhindert werden. Allerdings hatten die Ingenieure den Zustand des Gelandetseins so festgelegt, dass beide Fahrwerke mit einem vorgegebenen Anpressdruck auf der Fahrbahn aufliegen müssen.

Da an jenem Tag starker Regen und strenge Scherwinde herrschten, erreichte die Maschine trotz Bodenkontakt nicht den vorgesehenen Anpressdruck. Der Umkehrschub konnte nicht rechtzeitig aktiviert werden, und die Maschine rollte mit hoher Geschwindigkeit über die Landebahn hinaus.

Um systematisch aus einer Aufgabenbeschreibung die Kandidaten für die verschiedenen Elemente zu gewinnen, müssen wir uns fragen:

- Welche dinglichen und abstrakten Gegenstände werden bearbeitet, verändert oder tauchen in Kommunikationssituationen auf?
- Welche Eigenschaften zeichnen diese Gegenstände aus?
- Welche Beziehungen bestehen zwischen ihnen?
- Wie wird mit ihnen umgegangen?
- Welche Rollen treten auf, und für welche Handlungen sind sie verantwortlich?

Bei der Substantivanalyse werden Substantive als mögliche Kandidaten für Klassen, Attribute oder Assoziationen und Verben als Kandidaten für Methoden identifiziert. Konkrete Attributwerte oder Objekte interessieren bei der Erstellung des Klassenmodells eher weniger.

Beispiel 1.1.23

Betrachten wir die Fallstudie (Beispiel 1.1.2), so finden wir unter anderem uninteressante Substantive und konkrete Namen wie Abbildung, Blumenladen und Beschreibung. Wir stoßen jedoch auch auf Klassenkandidaten wie Pflanze, Dekorationsartikel und Kunde.

1.1.5 UML-Diagramme

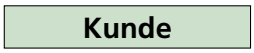
Wir werden nun zwei Typen von Diagrammen aus der Modellierungssprache UML einführen, die im weiteren Verlauf, insbesondere den Lektionen 2 und 3, zur visuellen Darstellung eingesetzt werden. Es gibt noch viele weitere UML-Diagramme, die in der Praxis Anwendung finden, aber kein Rolle in diesem Kurs spielen.

UML-Klassendiagramme Mit UML-Klassendiagrammen, deren Elemente wir im Folgenden vorstellen, können Klassen und ihre Beziehungen beschrieben werden.

Definition 1.1.24 : UML-Klassendiagramm

Ein UML-Klassendiagramm beschreibt grafisch die Attribute, Methoden sowie Assoziationen und Generalisierungen zwischen Klassen.

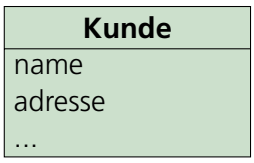
Eine Klasse wird wie in der folgenden Abbildung in einem UML-Klassendiagramm als unausgefülltes Rechteck dargestellt, wobei der Klassenname in das Rechteck geschrieben wird:



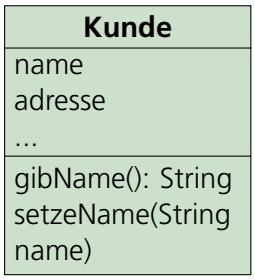
Um die Lesbarkeit zu erhöhen, verwenden wir in diesem Skript einen Farbcode für UML-Diagramme. Klassen werden grün, Objekte gelb und Methodenrahmen blau gezeichnet. Das ist aber eine Konvention nur für diesen Kurs, klassischerweise würden alle drei ohne Farbcodierung gezeichnet.

Um die Attribute einer Klasse in einem UML-Klassendiagramm darzustellen, wird dem Rechteck ein neuer Abschnitt hinzugefügt. In diesem Abschnitt wird ein Attribut pro Zeile aufgezählt. Das Attribut muss mindestens einen Namen aufweisen.

Hat eine Klasse keine Attribute oder sind sie in dem aktuellen Diagramm nicht von Interesse, so kann der gesamte Abschnitt entfallen. Werden nicht alle Attribute im Diagramm erwähnt, so kann das Vorhandensein weiterer Attribute mit drei Punkten am Ende des Abschnitts angedeutet werden.



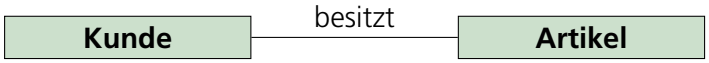
Ebenso kann optional ein Abschnitt für Methoden eingefügt werden. Dieser wird unterhalb des Abschnitts für Attribute ergänzt. Dabei wird auch eine Methode pro Zeile aufgezählt. Es muss mindestens der Name und in Klammern die (möglicherweise leere) Parameterliste genannt werden. Hat die Methode einen Rückgabetypen, so wird dieser durch einen Doppelpunkt getrennt hinter dem Methodennamen angegeben. Wie schon bei den Attributen kann das Vorhandensein weiterer Methoden mit drei Punkten am Ende des Abschnitts angedeutet werden.



Bemerkung 1.1.25

Weitere Eigenschaften von Attributen und Methoden und deren Notation in UML werden wir erst später im Kurs kennen lernen.

Eine Assoziation zwischen zwei Klassen wird als einfache Linie zwischen den Klassen angedeutet. Der Name der Assoziation wird an die Linie geschrieben. Solange keine einfachen Pfeilspitzen an den Enden angegeben oder die Enden mit einem Kreuz gekennzeichnet werden, ist die Navigierbarkeit der Assoziation undefiniert. Die Navigierbarkeit gibt an, welches der beteiligten Objekte einen Verweis auf das jeweils andere besitzt. Die folgende Grafik zeigt eine Assoziation *besitzt* ohne Angabe der Navigierbarkeit:



Die Navigierbarkeit kann wie folgt expliziert werden: Die Pfeilspitze besagt, dass eine Navigierbarkeit in die Richtung der Pfeilspitze ausdrücklich gegeben ist, das Kreuz besagt hingegen, dass eine Navigierbarkeit in die Richtung des Kreuzes ausdrücklich untersagt ist. Man kann auch, je nach Anwendungsfeld nur ein Kreuz oder einen Pfeil verwenden, oder an beide Seiten einen Pfeil schreiben.

Oft will man bei Assoziationen ausdrücken, mit wie vielen Objekten einer anderen Klasse ein Objekt in Beziehung stehen kann bzw. muss. Beispielsweise muss eine Rechnung immer für genau ein Kunde ausgestellt worden sein. Es kann aber durchaus mehrere Rechnungen für ein Kunde geben. Solche Aussagen werden in der UML durch Multiplizitäten dargestellt.

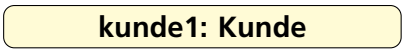
Um auszudrücken, dass eine Rechnung für genau ein Kunde ausgestellt wird, steht an dem Ende der Assoziation bei der Klasse Kunde eine 1. Die Aussage, dass es zu einem Kunden mehrere Rechnungen geben kann, wird durch ein $*$ an dem anderen Ende der Assoziation dargestellt. Ein $*$ steht dabei für beliebig viele.

Diese Beziehung kann also gelesen werden als ein Kunde kann Empfänger beliebig vieler Rechnungen sein (Leserichtung von Kunde zu Rechnung) und eine Rechnung muss exakt ein Kunde als Empfänger haben (Leserichtung von Rechnung zu Kunde).

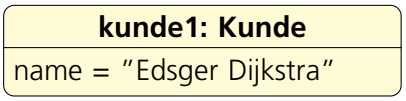
Eine gültige Multiplizität kann entweder eine ganze positive Zahl oder $*$ sein. Wenn ein Bereich angegeben werden soll, so wird dieser mit *untereGrenze*..*obereGrenze* dargestellt. Dabei kann *obereGrenze* auch wieder $*$ sein.

Eine Generalisierungsbeziehung wird mit unausgefüllten Pfeilspitzen dargestellt, wobei die Pfeilspitze zur allgemeineren Klasse zeigt.

UML-Objektdiagramme Um konkrete Situationen beschreiben zu können, ist es manchmal hilfreich Objekte darzustellen. In einem UML-Objektdiagramm wird ein Objekt durch ein Rechteck (meist mit abgerundeten Ecken) dargestellt. In dem Rechteck stehen, durch einen Doppelpunkt getrennt, der Name und die Klasse des Objekts. Hat das Objekt keinen speziellen Namen, so kann dieser auch entfallen.



Wenn gewünscht, kann auch noch ein Abschnitt für die Attributwerte ergänzt werden.



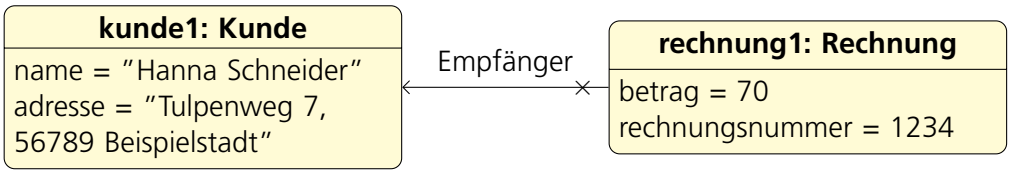
Verknüpfungen zwischen Objekten werden auf dieselbe Art dargestellt wie Assoziationen zwischen Klassen.

Selbsttestaufgabe 1.1.3

Stellen Sie die folgende Situation in Form eines UML-Objektdiagramms dar.

Die Rechnung über 70 Euro mit der Nummer 1234 ist am 17.8.1998 für die Kundin Hanna Schneider ausgestellt worden. Hanna Schneider wohnt im Tulpenweg 7 in 56789 Beispielstadt.

Musterlösung zu Selbsttestaufgabe 1.1.3



1.2 Mengen, Funktionen und Strukturen

In diesem Abschnitt werden wir mathematische Grundbegriffe wiederholen, die im Regelfall aus der Schule bekannt sein sollten und in diesem Kurs Verwendung finden. Wer sich in mathematischen Grundbegriffen sicher fühlt, kann diesen Abschnitt nur querlesen, ob wirklich alle Begriffe bekannt sind. Es sei außerdem an dieser Stelle gesagt, dass die Wiederholung dieser Grundbegriffe nur auf einem vereinfachten Niveau geschieht. Für eine saubere Aufbereitung der Themen bietet es sich an, auf Kurse aus dem mathematischen Angebot der FernUniversität zurückzugreifen.

Wir verwenden die folgenden Zeichen mit der entsprechenden Bedeutung hier und im gesamten Skript:

Notation	Bedeutung
$a < b$	a ist kleiner als b
$a \leq b$	a ist kleiner als b oder gleich b
$a > b$	a ist größer als b
$a \geq b$	a ist größer als b oder gleich b
$a = b$	a ist gleich b
$a \neq b$	a ist ungleich b
$a \Leftrightarrow b$	Aussage a gilt genau dann, wenn Aussage b gilt.
$a \Rightarrow b$	Wenn Aussage a gilt, dann gilt auch Aussage b.
$a \Leftarrow b$	Wenn Aussage b gilt, dann gilt auch Aussage a.
$\forall a : b$	Für alle a gilt b.
$\exists a : b$	Es gibt ein a für das b gilt.

1.2.1 Mengen

Wenngleich der Begriff der Menge allgegenwärtig ist, ist eine axiomatische Definition des Begriffs „Menge“ sehr kompliziert und würde an dieser Stelle zu weit führen. Daher schaffen wir zunächst ein intuitives Verständnis des Begriffs Menge:

Definition 1.2.1 : Menge

Eine Menge M ist eine Sammlung individuell unterscheidbarer Elemente. Für ein beliebiges Element e schreiben wir $e \in M$, wenn wir sagen wollen, dass e in der Menge M enthalten ist und $e \notin M$ anderenfalls. Die Mächtigkeit einer Menge M ist die Anzahl der Elemente in M, wir schreiben $|M|$. Enthält M endlich viele Elemente, so nennen wir M endlich, anderenfalls unendlich.

Wir können Mengen auf verschiedene Weisen notieren:

- Vollständige Aufzählung der Elemente: $M = \{a, b, c\}$. Das ist in der Form natürlich nur möglich, wenn M eine endliche Menge ist.
- Einschränkung der Elemente einer Obermenge M': $M = \{x \in M' \mid p(x)\}$ zu lesen als: M enthält alle Elemente x aus M', so dass gilt $p(x)$, wobei $p(x)$ irgendein Prädikat, d. h. irgendeine Aussage über x ist, die entweder wahr oder falsch sein kann.
- Angabe eines Musters: $M = \{a, b, c, ...\}$. Hierbei ist zu beachten, dass das Muster in jedem Fall sehr deutlich erkennbar sein muss, denn gibt man zu wenige Elemente an, gibt es oft mehrere einfache Passungen auf das Muster. Ein Beispiel hierfür wäre: $\{2, 3, 5, ...\}$ – charakterisiert das alle Primzahlen oder alle Zahlen, die keine Quadratzahlen sind? Es könnte sich auch um eine Auflistung aller nicht-trivialen Fibonacci-Zahlen handeln. Schon die Angabe einer einzigen weiteren Zahl unterscheidet diese drei Optionen eindeutig: 7 für die Primzahlen, 6 für die nicht-Quadratzahlen und 8 für die Fibonacci-Zahlen.

Bemerkung 1.2.2 : Widersprüchlichkeit der Definition der Menge

An dieser Stelle merken wir gleich an, dass die angegebene Definition einfach und schlüssig wirken mag, bei kritischer Untersuchung aber Paradoxien zulässt. Ein bekanntes Beispiel ist die „Menge M aller Mengen, die sich nicht selbst enthalten". Es stellt sich die Frage, ob $M \in M$ gilt oder nicht. Angenommen $M \in M$, dann ist, auf Grund der Definition von M, M eine Menge, die sich nicht selbst enthält und somit $M \notin M$, was ein Widerspruch zur Annahme darstellt, dass $M \in M$ gilt. Umgekehrt, wenn wir annehmen, dass gilt $M \notin M$, dann ist M also eine Menge, die sich nicht selbst enthält und nach Definition von M gilt somit $M \in M$, was ebenfalls einen Widerspruch erzeugt.

Für die Zwecke dieses Kurses umgehen wir derartige Widersprüche durch die Forderung, dass nur endliche Mengen ihrerseits Mengen als Elemente haben dürfen. Das ist in vielen mathematischen Kontexten eine zu starke Einschränkung, im Kontext dieses Kurses vermeiden wir damit aber logische Widersprüchlichkeit, ohne benötigte Ausdrucksmächtigkeit zu verlieren.

Beispiel 1.2.3

- $M_0 = \{\} = \emptyset$ die leere Menge. Wir werden das Zeichen $\emptyset$ im gesamten Skript verwenden, um diese Menge zu bezeichnen.
- $M_1 = \{1, 2, 3, 4, 5\}$ ist die Menge, die die Zahlen 1 bis 5 enthält.
- $M_2 = \{1, 2, 3, 4, 5, ...\} = \mathbb{N}$ ist die Menge aller natürlichen Zahlen
- $M_3 = \{n \in M_2 \mid \exists m \in M_2 : n = 2 \cdot m\} = \{n \in M_2 \mid$ Es gibt eine Zahl $m \in M_2$ so dass $n = 2 \cdot m\}$ ist die Menge aller geraden natürlichen Zahlen.

Die Beispiele verdeutlichen auch eine Beobachtung für Mengen, die manchem zunächst unintuitiv erscheint: Ein Element kann nur in einer Menge sein oder nicht, es macht also keinen Unterschied, „wie oft" ein Element einer Menge hinzugefügt wird. Die Mengen $\{1, 2, 3\}$ und $\{1, 2, 2, 3, 3, 3\}$ enthalten exakt die gleichen Elemente; es handelt sich nur um unterschiedliche Schreibweisen für die identische Menge.

Natürlich ist es auf Dauer müßig, jede Menge grundständig zu definieren, stattdessen werden wir immer wieder Mengenoperationen verwenden, um neue Mengen aus bestehenden Mengen zu bilden.

Definition 1.2.4 : Mengenoperationen

Es seien M_1, M_2 Mengen, dann sind:

- $M = M_1 \cup M_2$ die Menge aller Elemente, die in M_1 enthalten sind, zuzüglich aller Elemente, die in M_2 enthalten sind. M ist die Vereinigung von M_1 und M_2.
- $M = M_1 \cap M_2$ die Menge aller Elemente, die sowohl im M_1, als auch in M_2 enthalten sind. M ist der Schnitt von M_1 und M_2.
- $M = M_1 \setminus M_2$ die Menge aller Elemente, die in M_1 enthalten sind, aber nicht in M_2. M ist die Differenzmenge von M_1 und M_2.
- $M = M_1 \times M_2 = \{(m_1, m_2) \mid m_1 \in M_1, m_2 \in M_2\}$ ist die Menge aller (geordneten) Paare von Elementen aus M_1 und M_2.

Wir führen einige Berechnungen mit diesen Operatoren durch:

Beispiel 1.2.5

- $\{1,2,3,4,5\} \cup \{3,4,5,6,7\} = \{1,2,3,4,5,6,7\}$
- $\{1,2,3,4,5\} \cap \{3,4,5,6,7\} = \{3,4,5\}$
- $\{1,2,3,4,5\} \setminus \{3,4,5,6,7\} = \{1,2\}$
- $\{1,2,3\} \times \{3,4,5\} = \{(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,3),(3,4),(3,5)\}$

Als Merkhilfe, sofern Sie bereits mit den logischen Operatoren $\vee$ (oder) und $\wedge$ (und) vertraut sind, beachten Sie die optische und semantische Ähnlichkeit mit $\cup$ und $\cap$. Es gilt

$$M_1 \cup M_2 = \{m \mid m \in M_1 \vee m \in M_2\}$$

sowie

$$M_1 \cap M_2 = \{m \mid m \in M_1 \wedge m \in M_2\}.$$

Wir merken außerdem an, dass Vereinigung und Schnitt oftmals auch auf Sammlungen von Mengen angewendet werden:

Bemerkung 1.2.6

Es sei M eine Sammlung von Mengen, dann sind die verallgemeinerte Vereinigung und der verallgemeinerte Schnitt wie folgt definiert:

- $\bigcup M = \{x \mid \exists M' \in M : x \in M'\} = \{x \mid$ Es gibt eine Menge M' in M so dass $x \in M'\}$ – also ist $\bigcup M$ die Vereinigung aller Mengen in M.

- $\bigcap M = \{x \mid \forall M' \in M : x \in M'\} = \{x \mid$ Für alle Mengen M' in M gilt $x \in M'\}$ – also ist $\bigcap M$ der Schnitt aller Mengen in M.

Die verallgemeinerte Vereinigung und der verallgemeinerte Schnitt ist, auf Grund unserer Einschränkungen an Mengen um Widersprüche zu vermeiden, nicht zwingend für alle Wahlen von M wiederum eine Menge, wir können aber für alle im Kontext dieses Kurses vorkommenden Vereinigungen und Schnitte (ohne weitere Überprüfung) davon ausgehen, dass sich eine Menge aus diesen Operationen ergibt.

Eine Besonderheit ergibt sich noch bei der Mehrfachanwendung von $\times$:

Bemerkung 1.2.7 : n-Tupel

Wenn wir den $\times$-Operator mehrfach anwenden, müsste man eigentlich mit Klammern unterscheiden, welche Elemente zuerst gepaart werden. Seien beispielsweise M_1, M_2, M_3 Mengen, dann gilt formal:

$$(M_1 \times M_2) \times M_3 \neq M_1 \times (M_2 \times M_3)$$

Allerdings ist es in den allermeisten Anwendungen – und allen, die wir in diesem Kurs betrachten werden – unerheblich, in welcher Reihenfolge die Paarung vorgenommen wird, demnach schreiben wir für beides einfach:

$$M_1 \times M_2 \times M_3$$

und identifizieren Tupel wie $(1, (2, 3))$ und $((1, 2), 3)$ einfach als $(1, 2, 3)$. Wir nutzen außerdem die abkürzende Schreibweise M^n für $M \times M \times ... \times M$ mit insgesamt n Vorkommen von M. So ist beispielsweise $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

Wir schließen die Betrachtungen zu Mengen mit grundlegenden Vergleichsoperatoren für Mengen:

Definition 1.2.8 : Vergleichsoperatoren auf Mengen

Es seien M_1, M_2 Mengen.

- $M_1 = M_2$ gilt, falls jedes Element aus M_1 auch im M_2 ist und umgekehrt jedes Element aus M_2 auch in M_1 ist. Wir sagen M_1 ist gleich M_2.
- $M_1 \neq M_2$ gilt, falls es ein Element in M_1 gibt, das nicht in M_2 ist oder umgekehrt. Wir sagen M_1 und M_2 sind nicht gleich.
- $M_1 \subseteq M_2$ gilt, falls jedes Element aus M_1 auch in M_2 ist, wir sagen dann M_1 ist Teilmenge (oder gleich) M_2.
- $M_1 \supseteq M_2$ gilt genau dann wenn $M_2 \subseteq M_1$. Wir sagen dann, dass M_1 eine

Obermenge (oder gleich) M_2 ist.

- $M_1 \subset M_2$ gilt, falls jedes Element aus M_1 auch in M_2 ist und es mindestens ein Element in M_2 gibt, das nicht in M_1 liegt, wir sagen dann M_1 ist eine (echte) Teilmenge von M_2.
- $M_1 \supset M_2$ gilt genau dann wenn $M_2 \subset M_1$. Wir sagen dann, dass M_1 eine (echte) Obermenge von M_2 ist.

Für alle diese Operatoren nutzen wir auch die durchgestrichene Schreibweise um zu sagen, dass eine gewisse Relation nicht erfüllt ist, so heißt z. B. $M_1 \not\subseteq M_2$ dass $M_1 \subseteq M_2$ nicht zutreffend ist.

Hiermit können wir eine weitere Möglichkeit, neue Mengen zu konstruieren, die Potenzmenge, definieren:

Definition 1.2.9 : Potenzmenge

Es sei M eine Menge, dann ist die Potenzmenge $\mathcal{P}(M)$ die Menge aller Teilmengen von M:

$$\mathcal{P}(M) = \{M' \subseteq M\}$$

Beispiel 1.2.10

- $\{1,2,3\} \subseteq \{1,2,3,4,5\}$
- $\{1,2,3\} \subset \{1,2,3,4,5\}$
- $\{1,2,3\} \subseteq \{1,2,3\}$ aber $\{1,2,3\} \not\subset \{1,2,3\}$
- $\{1,2,3\} \supseteq \{1\}$
- $\{1,2,3\} \supset \{1\}$
- $\{1,2,3\} = \{3,1,2\}$
- $\{1,2,3\} \neq \{1,2\}$
- $\mathcal{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
- $\mathcal{P}(\emptyset) = \{\emptyset, \{\emptyset\}\}$

1.2.2 Funktionen, Gruppen, Körper

Eine zentrale Operation auf Mengen ist die Funktion oder Abbildung. Eine Funktion ordnet jedem Element aus ihrem Definitionsbereich ein Element aus dem Wertebereich zu. Im

Kontext der Berechenbarkeitstheorie und Komplexität (Lektion 6) werden wir zudem von den etwas schwächeren partiellen Funktionen Gebrauch machen, bei denen nicht zwingend für jedes Element aus dem Definitionsbereich ein Funktionswert zugeordnet werden muss.

Definition 1.2.11 : Funktion

Eine Funktion f besteht aus drei Komponenten, einer Menge D die wir den Definitionsbereich nennen, einer Menge W, die wir den Wertebereich nennen und einer Zuordnungsvorschrift. Formal schreiben wir eine Funktion auf eine der folgenden Weisen:

$$f\colon D \to W, f(x) = y \qquad f\colon D \to W, x \mapsto y$$

Dabei ist $x \in D$ und $y = f(x) \in W$.

Wenn f auf allen Werten des Definitionsbereichs definiert ist, nennen wir f eine totale Funktion oder kurz nur Funktion. Ist f nicht notwendigerweise für jeden Wert im Definitionsbereich definiert, so nennen wir f eine partielle Funktion. In den meisten Feldern der Mathematik betrachtet man nur totale Funktionen, weswegen es üblich ist, totale Funktionen nur kurz als Funktionen (oder Abbildungen) zu bezeichnen und immer dann, wenn eine Funktion nicht notwendigerweise total ist, stattdessen partielle Funktion zu schreiben. Ist eine partielle Funktion f auf einem Wert x nicht definiert, so schreiben wir gelegentlich auch $f(x) =$ undefiniert.

Wir betrachten einige Beispielsfunktionen:

Beispiel 1.2.12

- $f_1\colon \mathbb{N} \to \mathbb{N} : f_1(n) = n + 1$ ist die Nachfolgerfunktion. Jeder natürlichen Zahl wird ihr Nachfolger zugeordnet.
- $f_2\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N} : f_2(x, y) = x + y$ ist die Additionsfunktion. Zwei Zahlen x und y wird ihre Summe zugeordnet.
- $f_3\colon \mathbb{N} \to \{0, 1\} : f_3(x) = \begin{cases} 0 & \text{falls } x \text{ gerade ist} \\ 1 & \text{sonst} \end{cases}$ ist die Funktion, die jede natürliche Zahl auf ihren Rest bei Division durch 2 abbildet. Hier verwenden wir auch erstmals eine Schreibweise, die wir immer wieder verwenden werden, wenn wir Fallunterscheidungen vornehmen müssen. Hinter der großen geschweiften Klammer werden von oben nach unten die verschiedenen Fälle spezifiziert. In jeder Zeile steht zunächst das Ergebnis und dann der Fall, für den dieses Ergebnis verwendet werden soll. Der letzte Fall darf mit „sonst“ beschriftet sein und erfasst alle nicht zuvor erfassten Fälle. Sollten die vorherigen Fälle nicht disjunkt sein, d.h. es gibt Eingaben, für die mehr als ein Fall greift, dann muss der Wert der beiden Fälle an der Stelle übereinstimmen, anderenfalls ist eine solche

Fallunterscheidung *nicht wohldefiniert*.

- $f_4 \colon \{1,2,3,4,5\} \to \{1,2,3,4,5\}, f(1) = 2, f(2) = 4, f(3) = 3, f(4) = 1, f(5) = 1$ ist eine Funktion, deren Zuordnungsvorschrift wir über explizite Angabe jedes Funktionswertes angegeben haben.
- $f_5 \colon \{1,2,3,4,5\} \to \{1,2,3,4\}, f(x) = \begin{cases} x & \text{falls } x \in \{1,2,3,4\} \\ \text{undefiniert} & \text{sonst} \end{cases}$ ist eine *partielle* Funktion (die nicht total ist).

Funktionen können miteinander verknüpft werden, üblicherweise setzt man dafür voraus, dass der Wertebereich der zuerst ausgeführten Funktion gleich dem Definitionsbereich der zweiten Funktion ist. Manchmal wird die Funktionsverknüpfung aber auch dann zugelassen, wenn der Wertebereich der ersten Funktion eine echte Teilmenge des Definitionsbereichs der zweiten Funktion ist.

Definition 1.2.13 : Funktionsverknüpfung

Es seien $f \colon X \to Y$ und $g \colon Y \to Z$ Funktionen, dann ist die Verknüpfung (oder Komposition) $g \circ f \colon X \to Z$ definiert mit der Funktionsvorschrift $(g \circ f)(x) = g(f(x))$. Wenn f und g partiell sind, ist $g \circ f$ auch partiell und $(g \circ f)(x))$ ist undefiniert, falls $f(x)$ undefiniert ist oder $f(x)$ definiert ist, aber $g(f(x))$ undefiniert ist.

Wir führen zudem das Kreuzprodukt auf Funktionen ein: Es seien weiterhin $f \colon X \to Y$ und $g \colon X' \to Y'$ Funktionen, dann ist das Kreuzprodukt von f und g definiert als:

$$f \times g \colon X \times X' \to Y \times Y', (f \times g)(x, x') = (f(x), g(x'))$$

Wie schon bei Mengen unterscheiden wir im Regelfall nicht zwischen $f \times (g \times h)$ und $(f \times g) \times h$.

Beispiel 1.2.14

Wir setzen unser Beispiel 1.2.12 fort:

- $f_3 \circ f_1$ ist definiert als $f_3 \circ f_1 \colon \mathbb{N} \to \{0,1\}$ mit der Zuordnungsvorschrift

$$(f_3 \circ f_1)(x) = \begin{cases} 0 & \text{falls } x+1 \text{ gerade ist} \\ 1 & \text{sonst} \end{cases} = \begin{cases} 0 & \text{falls } x \text{ ungerade ist} \\ 1 & \text{sonst} \end{cases}$$

- $f_1 \circ f_3$ ist nicht definiert, denn der Wertebereich von f_3 ist $\{0,1\}$ und der Definitionsbereich von f_1 ist $\mathbb{N}$, es gilt aber $\mathbb{N} \neq \{0,1\}$.
- $f_6 = f_1 \times f_2 \colon \mathbb{N} \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \times \mathbb{N}$ ist definiert gemäß $f_6(x,y,z) = (f_1(x), f_2(y,z)) = (x+1, y+z)$

- $f_2 \circ f_6 \colon \mathbb{N} \times (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$ ist definiert gemäß

$$(f_2 \circ f_6)(x, y, z) = f_2(x + 1, y + z) = x + 1 + y + z$$

- Die Funktion f_1 kann beliebig oft mit sich selbst verknüpft werden, weil der Definitions- und Wertebereich übereinstimmen. Wir schreiben für die n-fache Hintereinanderausführung von f_1 kurz $f_1^n \colon \mathbb{N} \to \mathbb{N}$ mit $f_1^n(x) = x + n$. Es gilt $f_1^1 = f_1$ und $f_1^2 = f_1 \circ f_1$.

Funktionen $f \colon D \to W$ heißen weiterhin *injektiv*, falls keine zwei Werte im Definitionsbereich auf den gleichen Funktionswert abbilden, i. Z.: $\forall x, y \in D : f(x) = f(y) \Rightarrow x = y$. Die Funktion heißt *surjektiv*, wenn es für jeden Wert in W mindestens einen Wert in D gibt, der auf W abbildet, i. Z.:$\forall y \in W \exists x \in D : f(x) = y$. Eine Funktion, die sowohl injektiv als auch surjektiv ist, nennen wir *bijektiv*.

Totale Funktionen, bei denen Definitions- und Wertebereich übereinstimmen, nennen wir auch Operationen auf dem Definitionsbereich. Auf dieser Basis können wir einige der wichtigsten mathematischen Strukturen definieren, die wir im Verlauf des Kurses immer wieder verwenden werden:

Definition 1.2.15 : Monoid, Halbgruppe, Gruppe

Wir fixieren eine Menge M.

- Gegeben sei (totale) Funktion $* \colon M \times M \to M$ nennen wir $(M, *)$ eine Halbgruppe, falls $*$ assoziativ ist, das heißt für alle $a, b, c \in M$ gilt: $(a*b)*c = a*(b*c)$.
- Eine Halbgruppe $(M, *)$ heißt Monoid wenn es ein neutrales Element e bzgl. $*$ in M gibt, d.h. $a * e = e * a = a$ für alle $a \in M$.
- Ein Monoid $(M, *)$ ist eine Gruppe, falls zudem für jedes Element $a \in M$ ein inverses Element $a^{-1} \in M$ existiert, d.h. es muss gelten $a * a^{-1} = a^{-1} * a = e$.
- Eine Gruppe $(M, *)$ heißt abelsch oder kommutativ, falls das Kommutativgesetz gilt, das heißt für alle $a, b \in M$ ist $a * b = b * a$.

Beispiel 1.2.16

- Gegeben sei eine endliche Menge Σ, dann bezeichnet Σ^* die Menge aller Sequenzen über Σ. Beispielsweise ist mit $\Sigma = \{a, b\}$ die Menge

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, ...\}$$

  die zugehörige Menge an Sequenzen oder Wörtern über a, b. Wir verwenden hier und in dem gesamten Kursmaterial ϵ um die leere Sequenz zu bezeichnen. Wir bezeichnen außerdem mit $\cdot$ die Konkatenation von zwei Zeichenketten, also

mit $w \in \Sigma^*$, $v \in \Sigma^*$ ist $w \cdot v = wv$. Beispielsweise ist also $ab \cdot ba = abba$. Dann ist $(\Sigma^*, \cdot)$ ein Monoid. Es gilt nämlich für alle $w, u, v \in \Sigma^*$: $w \cdot (v \cdot u) = w \cdot (vu) = wvu = (wv) \cdot u = (w \cdot v) \cdot u$, also gilt das Assoziativgesetz für $\cdot$. Das neutrale Element bzgl. $\cdot$ ist ϵ, denn $\epsilon \cdot w = w = w \cdot \epsilon$ für alle Wörter $w \in \Sigma^*$. Da kein Wort außer ϵ selbst so fortgesetzt werden kann, dass es das leere Wort ergibt, handelt es sich aber nicht um eine Gruppe – es gibt nämlich keine inversen Elemente.

- Die Menge $\{0, 1\}$ zusammen mit der Operation $\wedge$, die gemäß der folgenden Zuordnungsvorschrift definiert ist:

$\wedge$	0	1
0	0	0
1	0	1

  ist ein Monoid mit neutralem Element 1, aber keine Gruppe, denn 0 hat kein inverses Element. Die Assoziativität kann auf Grund der wenigen Elemente durch einfaches Ausprobieren aller Kombinationen nachgewiesen werden. Wir nennen die Operation $\wedge$ auch „und“.

- Die Menge $\{0, 1\}$ zusammen mit der Operation $\vee$, die gemäß der folgenden Zuordnungsvorschrift definiert ist:

$\vee$	0	1
0	0	1
1	1	1

  ist ein Monoid mit neutralem Element 0, aber keine Gruppe, denn 1 hat kein inverses Element. Wir nennen die Operation $\vee$ auch „oder“.

- Die Menge $\{0, 1\}$ zusammen mit der Operation $+$, die gemäß der folgenden Zuordnungsvorschrift definiert ist:

$+$	0	1
0	0	1
1	1	0

  ist ein Monoid mit neutralem Element 0, und auch eine Gruppe: das inverse Element von 0 ist 0 und das inverse Element von 1 ist 1. Zudem ist $(\{0, 1\}, +)$ auch kommutativ. Wir nennen die Operation $+$ auch Addition (im $\mathbb{Z}_2$) oder XOR.

- Die Menge $\mathbb{N}$ zusammen mit der Addition $+$ ist eine Halbgruppe, aber kein Monoid, denn es gibt kein neutrales Element bzgl. der Addition. Erweitert man die betrachtete Menge um die 0, so erhält man den Monoid $(\mathbb{N}_0, +)$ – es handelt sich hier aber nicht um eine Gruppe, weil die additiven Inversen (für alle Zahlen außer 0) fehlen. Hingegen ist $\mathbb{Z}$ zusammen mit $+$ eine kommutative Gruppe.

Die Assoziativität und Kommutativität von $+$ wird an dieser Stelle nicht bewiesen.

In vielen Fällen sind wir aber nicht nur an Mengen mit einer, sondern mit zwei Operationen interessiert – einer Addition und einer Multiplikation.

Definition 1.2.17 : Ringe, Körper

- Eine Menge M mit zwei Operationen $\oplus$ und $\otimes$ heißt Halbring oder Semiring, falls $(M, \oplus)$ ein kommutativer Monoid ist, dessen neutrales Element wir als 0 bezeichnen. Zusätzlich muss $(M, \otimes)$ eine Halbgruppe sein und das Distributivgesetz gelten, also für alle $a, b, c \in M$ gilt:

  $$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \qquad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

  Ist $(M, \otimes)$ ein Monoid, so nennen wir M unitär oder einen Halbring mit 1.

- $(M, \oplus, \otimes)$ ein Ring, falls $(M, \oplus, \otimes)$ ein Halbring ist und zusätzlich $(M, \oplus)$ eine kommutative Gruppe ist. Ist $(M, \otimes)$ ein Monoid, so nennen wir M unitär oder einen Ring mit 1.

- $(M, \oplus, \otimes)$ ist ein Körper, falls $(M, \oplus, \otimes)$ ein Ring ist und zusätzlich $(M \setminus \{0\}, \otimes)$ eine kommutative Gruppe ist.

Beispiel 1.2.18

- Die sogenannte Boolesche Algebra $(\{0, 1\}, \vee, \wedge)$ ist ein Halbring. Wir haben bereits besprochen, dass $(\{0, 1\}, \vee)$ und $(\{0, 1\}, \wedge)$ Halbgruppen sind, an der Symmetrie der Zuordnungsvorschrift sieht man auch, dass beide kommutativ sind und durch Abgleichen aller acht Wertkombinationen sieht man auch ein, dass das Distributivgesetz gilt.

- $(\mathbb{N}_0, +, \cdot)$, wobei $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, ist ein Halbring, aber kein Ring, denn nicht jedes Element hat ein additives Inverses in $\mathbb{N}$. Damit $a + b = 0$ mit $a, b \in \mathbb{N}_0$ gelten kann, muss $a = b = 0$ gelten.

- $(\mathbb{Z}, +, \cdot)$, wobei $\mathbb{Z} = \mathbb{N}_0 \cup \{x \mid -x \in \mathbb{N}\}$ die ganzen Zahlen sind, ist ein Ring, aber kein Körper, denn nicht jedes Element hat ein multiplikatives Inverses in $\mathbb{Z}$. Damit $a \cdot b = 1$ mit $a, b, \in \mathbb{Z}$ gelten kann, muss $a = b \in \{1, -1\}$ gelten.

- $(\mathbb{Q}, +, \cdot)$, wobei $\mathbb{Q} = \{\frac{a}{b} \mid a \in \mathbb{Z}, b \in \mathbb{N}\}$ die rationalen Zahlen sind, ist ein Körper.

- $(\mathbb{R}, +, \cdot)$, wobei $\mathbb{R}$ die reelen Zahlen Zahlen sind, ist ein Körper. $\mathbb{R}$ kann beispielsweise verstanden werden als der (einzige) vollständig angeordnete Körper, der $\mathbb{Q}$ enthält. Die Definition der reellen Zahlen müssen Sie für diesen Kurs nicht vollständig verstehen, Sie sollten aber wissen, dass die reellen Zahlen es erlauben, die Wurzel einer jeden nicht-negativen Zahl zu ziehen.

In unseren Beispielen werden wir immer mal wieder mit Matrizen und Vektoren zu tun bekommen. Wir können an dieser Stelle natürlich keine Einführung in die Theorie der Vektorräume geben, wir erinnern aber kurz an die wesentlichen Operationen, die in unseren Anwendungsbeispielen eine Rolle spielen. Wenn Sie mit diesen Begriffen nicht vertraut sind, werden Sie für den Rahmen dieses Kurses mit den folgenden Definitionen auskommen, es wäre aber empfehlenswert, sich mit den Konzepten vertraut zu machen. Hierzu kann zum Beispiel das Buch „Lineare Algebra: Eine Einführung für Studienanfänger“ von Gerd Fischer, beispielsweise in der 18. Auflage von 2014 im Springer-Verlag verwendet werden.

Definition 1.2.19

Wir betrachten für ein beliebiges $n \in \mathbb{N}$ den Vektorraum $(\mathbb{R}^n, +, \cdot)$ aller n-dimensionalen Vektoren über $\mathbb{R}$. Es gilt $\mathbb{R}^n = \{(x_1, x_2, ..., x_n) \mid x_1, x_2, ..., x_n \in \mathbb{R}\}$. Die Vektoraddition $+$ ist komponentenweise definiert, also für beliebige $(x_1, x_2, ..., x_n), (y_1, y_2, ..., y_n) \in \mathbb{R}^n$ gilt

$$(x_1, x_2, ..., x_n) + (y_1, y_2, ..., y_n) = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n)$$

Die Skalarmultiplikation $\cdot$ ist definiert wie folgt: Gegeben einen beliebigen Vektor $(x_1, x_2, ..., x_n) \in \mathbb{R}$ und einen beliebigen Skalar, das heißt eine beliebige Zahl $a \in \mathbb{R}$, dann gilt

$$a \cdot (x_1, x_2, ..., x_n) = (a \cdot x_1, a \cdot x_2, ..., a \cdot x_n)$$

Diese Vektorräume sind für uns vor allem interessant, weil Matrizen als spezielle Funktionen, sogenannte lineare Abbildungen, zwischen diesen Räumen vermitteln. Wenn wir Matrizenrechnung betrachten, schreiben wir die Vektoren des $(\mathbb{R}^n, +, \cdot)$ als Spaltenvektoren, also $(x_1, x_2, ..., x_n)$ wird dann geschrieben als

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Eine Matrix $m \in \mathbb{R}^{x \times y}$ bildet einen Vektor des $\mathbb{R}^y$ auf einen Vektor des $\mathbb{R}^x$ ab. Eine Matrix operiert auf einem Vektor mittels der Matrixmultiplikation. Um die Multiplikation einer Matrix mit einem Vektor formal beschreiben zu können, benötigen wir die Summenschreibweise:

Definition 1.2.20 : Summenschreibweise

Das Summenzeichen $\sum$ kann verwendet werden, um mehrgliedrige Summen auszudrücken. Hierzu gibt es zwei besonders übliche Nutzungsweisen. Wir definieren die Schreibweisen für die reellen Zahlen, aber grundsätzlich kann die Schreibweise immer verwendet werden, wenn man einen beliebigen Zahlenraum mit einer Additionsoperation verwendet. Gegeben sei eine Menge M von Zahlen, dann bedeutet $\sum M$, dass man die Summe aller Elemente von M bildet. Ist M beispielsweise endlich, kann also geschrieben werden als $M = \{m_1, m_2, ..., m_n\}$, dann ist $\sum M = m_1 + m_2 + ... + m_n$. Alternativ kann auch ein Zählindex verwendet werden. Sei $a(i)$ ein Ausdruck, der für alle Werte $1 \leq i \leq n$ definiert ist, dann ist beispielsweise

$$\sum_{i=1}^{n} a(i) = a(1) + a(2) + ... + a(n)$$

Analog definieren wir, falls die anzuwendende Operation $\cdot$ statt $+$ heißt, das Produktzeichen $\prod$ um mehrgliedrige Produkte auszudrücken als $\prod M$ für das Produkt aller Elemente von M bzw. $\prod_{i=1}^{n} a(i) = a(1) \cdot a(2) \cdot ... \cdot a(n)$.

Hiermit können wir jetzt die Multiplikation einer Matrix $m \in \mathbb{R}^{x \times y}$ mit einem Vektor $v \in R^y$ wie folgt definieren:

$$m \cdot v = \begin{pmatrix} m_{1,1} & m_{1,2} & ... & m_{1,y} \\ m_{2,1} & m_{2,2} & ... & m_{2,y} \\ \vdots & \vdots & \ddots & \vdots \\ m_{x,1} & m_{x,2} & ... & m_{x,y} \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_y \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{y} (m_{1,i} \cdot v_i) \\ \sum_{i=1}^{y} (m_{2,i} \cdot v_i) \\ \vdots \\ \sum_{i=1}^{y} (m_{x,i} \cdot v_i) \end{pmatrix}$$

Zur Illustration dieser Rechenregel führen wir eine Matrix-Vektor-Multiplikation durch:

Beispiel 1.2.21 : Matrix-Vektor-Multiplikation

$$\begin{pmatrix} 2 & 3 & 4 \\ 5 & 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 2 \cdot 1 + 3 \cdot 3 + 4 \cdot 5 \\ 5 \cdot 1 + 2 \cdot 3 + 2 \cdot 5 \end{pmatrix} = \begin{pmatrix} 31 \\ 21 \end{pmatrix}$$

Schließlich kann man auch zwei Matrizen $x \in \mathbb{R}^{k \times m}$, $y \in \mathbb{R}^{m \times n}$ multiplizieren, um eine Matrix $z \in \mathbb{R}^{k \times n}$ zu erhalten. Der Eintrag $z_{i,j}$ ergibt sich jeweils als Produkt der i-ten Zeile

von x und der j-ten Spalte von y:

$$x \cdot y = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k,1} & x_{k,2} & \dots & x_{k,m} \end{pmatrix} \cdot \begin{pmatrix} y_{1,1} & y_{1,2} & \dots & y_{1,n} \\ y_{2,1} & y_{2,2} & \dots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \dots & y_{m,n} \end{pmatrix}$$
$$= \begin{pmatrix} \sum_{i=1}^{m}(x_{1,i} \cdot y_{i,1}) & \sum_{i=1}^{m}(x_{1,i} \cdot y_{i,2}) & \dots & \sum_{i=1}^{m}(x_{1,i} \cdot y_{i,n}) \\ \sum_{i=1}^{m}(x_{2,i} \cdot y_{i,1}) & \sum_{i=1}^{m}(x_{2,i} \cdot y_{i,2}) & \dots & \sum_{i=1}^{m}(x_{2,i} \cdot y_{i,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{m}(x_{k,i} \cdot y_{i,1}) & \sum_{i=1}^{m}(x_{k,i} \cdot y_{i,2}) & \dots & \sum_{i=1}^{m}(x_{k,i} \cdot y_{i,n}) \end{pmatrix}$$

Führen wir einmal eine Beispiels-Matrix-Multiplikation aus:

Beispiel 1.2.22 : Matrix-Multiplikation

$$\begin{pmatrix} 2 & 3 & 4 \\ 5 & 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 1 \\ 5 & 3 \end{pmatrix} = \begin{pmatrix} 2 \cdot 1 + 3 \cdot 3 + 4 \cdot 5 & 2 \cdot 2 + 3 \cdot 1 + 4 \cdot 3 \\ 5 \cdot 1 + 2 \cdot 3 + 2 \cdot 5 & 5 \cdot 2 + 2 \cdot 1 + 2 \cdot 3 \end{pmatrix} = \begin{pmatrix} 31 & 19 \\ 21 & 18 \end{pmatrix}$$

Die Matrix-Vektor-Multiplikation kann so auch verstanden werden als spezielle Matrix-Matrix-Multiplikation, wobei der Vektor als $\mathbb{R}^{n \times 1}$-Matrix aufgefasst wird.

Wir schließen die Betrachtung von Funktionen noch mit einigen speziellen Funktionen, die im Verlauf des Kurses Verwendung finden:

Beispiel 1.2.23 : Spezielle Funktionen

- Die Funktion max: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ mit

$$\max(x, y) = \begin{cases} x & \text{falls } x > y \\ y & \text{sonst} \end{cases}$$

  Diese Funktion wird auch für Zahlenräume, die in $\mathbb{R}$ enthalten sind, eingeschränkt, d.h. mit passend verkleinertem Definitions- und Wertebereich aber identischer Zuordnungsvorschrift, verwendet.

- Die Funktion min: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ mit

$$\min(x, y) = \begin{cases} x & \text{falls } x < y \\ y & \text{sonst} \end{cases}$$

  Diese Funktion wird auch für Zahlenräume, die in $\mathbb{R}$ enthalten sind, eingeschränkt verwendet.

- Sowohl min als auch max werden auch auf nicht-leere endliche Mengen M verallgemeinert, dabei gilt:

  $$\mathsf{max}(\{m\}) = m \qquad \mathsf{max}(\{m_1, m_2, ..., m_n\}) = \mathsf{max}(m_1, \mathsf{max}(\{m_2, ..., m_n\}))$$

  und analog für min. Gelegentlich werden die Funktionen sogar auf unendliche Mengen angewendet, hier muss man aber beachten, dass die Funktionen dann partiell werden: Bei unendlichen Mengen ist das Maximum und das Minimum nicht immer definiert.

- Die Funktion $\lceil\cdot\rceil : \mathbb{R} \to \mathbb{Z}$ mit

  $$\lceil x \rceil = \mathsf{min}(\{z \in \mathbb{Z} \mid z \geq x\})$$

  ergibt die nächstgrößere ganze Zahl (Aufrunden). Das Minimum ist hier immer definiert, weil die betrachtete Menge nach unten beschränkt ist.

- Die Funktion $\lfloor\cdot\rfloor : \mathbb{R} \to \mathbb{Z}$ mit

  $$\lfloor x \rfloor = \mathsf{max}(\{z \in \mathbb{Z} \mid z \leq x\})$$

  ergibt die nächstkleinere ganze Zahl (Abrunden). Das Maximum ist hier immer definiert, weil die betrachtete Menge nach oben beschränkt ist.

- Die Funktion div: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

  $$\mathsf{div}(x, y) = \left\lfloor \frac{x}{y} \right\rfloor$$

  ist die Division ohne Rest.

- Die Funktion mod: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

  $$\mathsf{mod}(x, y) = x - y \cdot (x \ \mathsf{div}\ y)$$

  ist der Rest der Division.

- Die Funktion $|\cdot| : \mathbb{R} \to \mathbb{R}$ mit

  $$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

  ist der Betrag einer Zahl.

- Die Logarithmusfunktion $\log\colon \mathbb{R}^{\geq 0} \to \mathbb{R}$ ist die Umkehrfunktion der Potenzfunktion:
$$\log(x) = y \Leftrightarrow 2^y = x$$
In manchen Fällen benötigen wir die Logarithmusfunktion auch für eine andere Basis, dann geben wir diese als Index mit an, es gilt also
$$\log_b(x) = y \Leftrightarrow b^y = x$$

- Als *Polynom* bezeichnen wir jede Funktion $p\colon \mathbb{R} \to \mathbb{R}$ mit einer Funktionsvorschrift der Form
$$p(x) = \sum_{i=0}^{n} a_i \cdot x^i$$

Mit diesen Funktionen können wir auch Primzahlen definieren, zur Erinnerung, eine Primzahl ist eine natürliche Zahl, die genau zwei Teiler hat: sich selbst und 1. Formal unter zu Hilfenahme der obigen Definitionen:

$$\mathsf{Prim} = \{n \in \mathbb{N} \mid |\{m \in \mathbb{N} \mid n \bmod m = 0\}| = 2\}$$

1.2.3 Relationen und Aussagenlogik

Relationen sind besondere Mengen, nämlich Teilmengen auf dem Kreuzprodukt einer Menge mit sich selbst. Mit Relationen können Elemente einer Menge miteinander in ein Verhältnis gestellt werden.

Definition 1.2.24 : Relation

Gegeben sei eine Menge M, dann ist jede Teilmenge $R \subseteq M \times M$ eine Relation. Wir schreiben für $x, y \in M$ für $(x, y) \in R$ gelegentlich auch $x\,R\,y$.

Beispiel 1.2.25 : Relationen

- Die wohl bekannteste Relation ist die Gleichheit auf einer beliebigen Menge M: $=\subseteq M \times M$ wobei die Gleichheitsrelation auf M definiert ist als $\{(m, m) \mid m \in M\}$.
- Auch $<$ ist eine Relation auf (zum Beispiel) $\mathbb{N}$ mit der Definition
$$\{(m, n) \in \mathbb{N} \times \mathbb{N} \mid \exists k \in \mathbb{N} : m + k = n\}$$
- Die Funktion mod kann verwendet werden um eine Relation auf $\mathbb{N}$ zu definieren:
$$\equiv_n = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \bmod n = y \bmod n\}$$

Wir schreiben dann auch $x \equiv y (\mathsf{mod}\ n)$, also x ist kongruent zu y modulo n.

- Eine Relation kann aber auch einfach durch Aufzählung der Paare definiert werden, beispielsweise ist $\{(1,2),(4,3)\}$ eine Relation auf $\mathbb{N}$.

Für Relationen gibt es eine Reihe von Eigenschaften, mit denen wir auch spezielle Relationstypen definieren können:

Definition 1.2.26 : Relationseigenschaften

Es sei $R \subseteq M \times M$ eine Relation, dann ist R:

- reflexiv falls für alle $m \in M$ gilt: $(m,m) \in R$.
- symmetrisch falls für alle $(m_1,m_2) \in R$ gilt $(m_2,m_1) \in R$.
- transitiv falls für alle $(m_1,m_2) \in R$ und $(m_2,m_3) \in R$ gilt $(m_1,m_3) \in R$.
- antisymmetrisch falls für alle $(m_1,m_2) \in R$ und $(m_2,m_1) \in R$ gilt $m_1 = m_2$.

Wir nennen eine Relation R eine *Äquivalenzrelation*, falls sie reflexiv, transitiv und symmetrisch ist und wir nennen sie eine *partielle Ordnung*, falls sie reflexiv, transitiv und antisymmetrisch ist.

Die einzige Relation auf einer Menge M, die sowohl eine Äquivalenzrelation, als auch eine partielle Ordnung ist, ist die Gleichheitsrelation. Die Äquivalenzrelation kann man als eine verallgemeinerte Gleichheitsrelation betrachten; beispielsweise ist die Relation $\equiv_n$, die wir oben definiert haben, eine sehr bekannte Äquivalenzrelation. Alle drei notwendigen Eigenschaften hierfür können durch einfachen Abgleich der Definitionen sofort eingesehen werden.

Schließlich kommen wir noch einmal auf die Boolesche Algebra $(\{0,1\}, \vee, \wedge)$ zurück. Wenn man 0 als false und 1 als true liest, dann entspricht $\vee$ dem C#-Operator | und $\wedge$ dem C#-Operator $\&$. Neben diesen beiden zweistelligen Operatoren betrachtet man oft noch die Implikation, die Biimplikation, die ebenfalls zweistellige Operatoren sind (also zwei logische Werte verknüpfen), und die Negation (die nur einen logischen Wert entgegen nimmt und damit einstellig ist):

$\rightarrow$	0	1
0	1	1
1	0	1

$\leftrightarrow$	0	1
0	1	0
1	0	1

x	0	1
$\neg x$	1	0

Es ist zu beachten, dass sich verschiedene logische Operatoren auch durch andere logische Operatoren ausdrücken lassen. So kann durch Abgleich der definierenden Tabellen leicht verifizieren, dass gilt:

$$a \rightarrow b \equiv \neg a \vee b \qquad a \leftrightarrow b \equiv (a \rightarrow b) \wedge (b \rightarrow a)$$

Wir schließen, indem wir an einige Umformungsregeln für aussagenlogische Formeln erinnern, die in Lektion 6 eine (kleine) Rolle spielen werden:

- Aborptionsregel: $a \wedge (a \vee b) \equiv a$
- deMorgans Gesetze: $\neg(a \wedge b) \equiv \neg a \vee \neg b$ und $\neg(a \vee b) \equiv \neg a \wedge \neg b$
- Idempotenz: $a \vee a \equiv a \wedge a \equiv a$
- Doppelte Negation: $\neg\neg a \equiv a$

1.3 Grundlegende Beweistechniken

Da wir, insbesondere in der hinteren Hälfte des Kursmaterials, gelegentlich Beweise führen werden, führen wir, jeweils begleitet mit einem Beispiel, drei grundlegende Beweistechniken ein, die in diesem und in anderen Kursen hilfreich sein können.

1.3.1 Induktion

Die Induktion ist in ihrer Grundform, der vollständigen Induktion, ein einfaches, aber sehr mächtiges Beweis-Instrument und üblicherweise die erste Beweistechnik, die in der Mathematikausbildung vermittelt wird. Die Induktion dient dazu, eine Aussage für alle natürlichen Zahlen zu beweisen:

Definition 1.3.1 : Vollständige Induktion

Gegeben sei eine Familie von Aussagen $a(i)$ für alle $i \in \mathbb{N}_0$, das heißt es gibt für alle natürlichen Zahlen i eine Aussage $a(i)$. Dann kann man beweisen, dass a für alle natürlichen Zahlen ab einem Startwert $n_0 \geq 0$ wahr ist wie folgt:

- **Induktionsanfang**: Zeige, dass $a(n_0)$ wahr ist.
- **Induktionsvoraussetzung**: Wir nehmen für ein $n \in \mathbb{N}_0$ an, dass $a(m)$ wahr ist, für alle $n_0 \leq m \leq n$.
- **Induktionsschritt**: Wir zeigen, dass dann gilt: $a(n+1)$ ist wahr.

Wieso funktioniert diese Beweistechnik? Wählen wir ein beliebiges $n \in \mathbb{N}_0$, das größer oder gleich n_0 ist. Dann können wir einsehen, dass mit einem erfolgreichen Induktionsbeweis $a(n)$ gelten muss wie folgt: Falls $n = n_0$ gilt, beweist der Induktionsanfang bereits $a(n)$. Anderenfalls verwenden wir zunächst den Induktionsanfang und wenden dann $n - n_0$ Mal den Induktionsschritt an. Auf diese Weise werden die Aussagen $a(n_0), a(n_0+1), ..., a(n_0+n-n_0) = a(n)$ nacheinander beweisen.

Als Beispiel zur Induktion beweisen wir, dass die folgende C#-Methode:

```
int f(int n)
{
    return n*(n+1)/2;
}
```

für alle Zahlen $n \in \mathbb{N}$ den Wert $\sum_{i=1}^{n} i$ zurückgibt.

- Induktionsanfang: Für $n = 1$ ergibt die Methode den Wert $1 \cdot (1+1)/2 = 1 \cdot \frac{2}{2} = 1 = \sum_{i=1}^{1} i$
- Induktionsvoraussetzung: Für alle $m \leq n$ ergebe die Methode den Wert $\sum_{i=1}^{m} i$
- Induktionsschritt: Für den Wert $n+1$ ergibt die Methode den Wert

  $$(n+1)\cdot(n+1+1)/2 = (n+1)\cdot(n+2)/2 = (n+1)\cdot n/2+(n+1)\cdot 2/2 = \sum_{i=1}^{n} i+(n+1)\cdot 1 = \sum_{i=1}^{n+1} i$$

  Im vorletzten Schritt haben wir die Induktionsvoraussetzung angewendet, indem wir den Teilausdruck $(n+1) \cdot n/2$ durch $\sum_{i=1}^{n} i$ ersetzt haben.

Aber Achtung: Dieser Beweis ignoriert, dass Variablen in C# einen beschränkten Werteraum haben. In vielen Fällen möchte man für seine Beweise über Software eine solche vereinfachte Annahme zulassen, allerdings ist die Aussage damit nicht mehr in jedem Fall richtig. Es ist sogar so, dass diese Methode nicht immer den richtigen Wert zurückgibt, wenn $\sum_{i=1}^{n} i$ selbst noch ein zulässiger int-Wert ist. Allerdings kann man die Methode einfach so anpassen, dass zumindest immer dann, wenn $\sum_{i=1}^{n} i$ ein int-Wert ist, das korrekte Ergebnis herauskommt:

```
int f(int n)
{
    if(n%2==0) return (n/2)*(n+1);
    else return n*((n+1)/2);
}
```

Selbsttestaufgabe 1.3.1 : Taubenschlagprinzip

Beweisen Sie das (endliche) Taubenschlagprinzip. Das endliche Taubenschlagprinzip lautet: Gegeben zwei endliche Mengen A und B mit $|A| > |B|$, dann gilt für jede (totale) Funktion $f\colon A \to B$, dass es mindestens ein Element $b \in B$ geben muss, so dass es zwei verschiedene Elemente $a_1, a_2 \in A$ gibt mit $f(a_1) = f(a_2) = b$; f ist also nicht injektiv. Dieser Beweis kann induktiv erbracht werden.

Das Taubenschlagprinzip hat seinen Namen aus der Taubenzucht. Taubenzüchter haben häufig einen Taubenschlag mit mehreren Sitzen. Das Taubenschlagprinzip besagt in diesem Bild: Immer wenn man mehr Tauben als Sitze im Taubenschlag hat, können nur dann alle Tauben Platz im Taubenschlag nehmen, wenn jedenfalls auf

einem Sitz mehr als eine Taube sitzt.

Musterlösung zu Selbsttestaufgabe 1.3.1

Wir zeigen das per Induktion über $|B|$.

Induktionsanfang ($|B| = 0$): In diesem Fall gibt es überhaupt keine Funktion $f\colon A \to B$, da A mindestens ein Element enthält, das auf kein Element von B abgebildet werden kann.

Induktionsvoraussetzung: Für alle $|A| > |B| \leq n \in \mathbb{N}_0$ gilt: Alle totalen Funktionen $f\colon A \to B$ sind nicht injektiv.

Induktionsschritt: Wir zeigen, dass für alle $|A| > |B| = n+1$ gilt, dass alle totalen Funktionen $f\colon A \to B$ nicht injektiv sind. Es sei hierzu $f\colon A \to B$ eine beliebige Funktion mit $|A| > |B| = n + 1$. Dann wählen wir ein beliebiges $b \in B$ aus. Falls es zwei oder mehr verschiedene Elemente $a_1, a_2 \in A$ gibt, so dass $f(a_1) = f(a_2) = b$, dann ist f nicht injektiv. Falls es genau ein Element $a \in A$ mit $f(a) = b$ gibt, setzen wir $A' = A \setminus \{a\}$, anderenfalls, falls also gar kein $a \in A$ existiert mit $f(a) = b$, setzen wir $A' = A$. Weiterhin setzen wir $B' = B \setminus \{b\}$. Dann können wir eine Funktion $f'\colon A' \to B'$ definieren gemäß $f'(a) = f(a)$ für alle $a \in A'$. Das ist dann eine Funktion mit $|A'| > |B'| = n$. Nach Induktionsvoraussetzung ist f' nicht injektiv, also gibt es mindestens ein $b' \in B'$ so dass $a_1, a_2 \in A'$ existieren mit $f'(a_1) = f'(a_2) = b$, aber $a_1 \neq a_2$. Also ist nach Definition von f' auch $f(a_1) = f(a_2)$ und f demnach nicht injektiv.

1.3.2 Widerspruchsbeweis

Ein Widerspruchsbeweis funktioniert nach dem sogenannten Prinzip des ausgeschlossenen Dritten. Um eine Aussage A zu beweisen, nehmen wir zunächst an, dass das Gegenteil der Aussage $\neg A$ der Wahrheit entspricht und zeigen dann, dass diese Annahme zu einem logischen Widerspruch führt. Da in der Realität aber kein logischer Widerspruch Bestand haben kann, muss $\neg A$ falsch und somit A wahr sein.

Definition 1.3.2 : Widerspruchsbeweis

Zu zeigen: Die Aussage A.
Vorgehen: Nimm an, dass $\neg A$ wahr ist und führe ausgehend von dieser Annahme logische Schlüsse unter Zuhilfenahme ausschließlich als wahr bekannter Aussagen einen logischen Widerspruch herbei.

Als Beispiel für einen Widerspruchsbeweis beweisen wir eine aus der Schulmathematik bekannte Aussage, nämlich dass $\sqrt{2}$ keine rationale Zahl ist. Für die Informatik hat diese Aussage zur Folge, dass es keine exakte Repräsentation von $\sqrt{2}$ auf dem Computer (und somit in C#) geben kann.

Angenommen, $\sqrt{2}$ sei eine rationale Zahl, dann kann $\sqrt{2} = \frac{p}{q}$ geschrieben werden mit $p \in \mathbb{Z}, q \in \mathbb{N}$. Wir können aber noch ein wenig mehr annehmen: $p \in \mathbb{N}$, denn wenn p negativ ist, dann ist $\frac{-p}{q} \cdot \frac{-p}{q} = \frac{(-p)\cdot(-p)}{q \cdot q} = \frac{p \cdot p}{q \cdot q} = 2$. Weiterhin können wir annehmen, dass $\mathsf{ggT}(p, q) = 1$; also $\frac{p}{q}$ ist gekürzt, denn falls $\mathsf{ggT}(p, q) > 1$, dann können wir statt p und q die Zahlen $p' := \frac{p}{\mathsf{ggT}(p,q)}$ und $q' := \frac{q}{\mathsf{ggT}(p,q)}$ verwenden und es gölte:

$$\frac{p'}{q'} = \frac{\frac{p}{\mathsf{ggT}(p,q)}}{\frac{q}{\mathsf{ggT}(p,q)}} = \frac{p}{\mathsf{ggT}(p, q)} \cdot \frac{\mathsf{ggT}(p, q)}{q} = \frac{p}{q}$$

Es würde also folgen, dass $\frac{p'}{q'} = \sqrt{2}$ und $\mathsf{ggT}(p', q') = 1$

Nun können wir rechnen:

$$\begin{aligned} &\frac{p}{q} \cdot \frac{p}{q} = 2 \\ \Leftrightarrow &\frac{p^2}{q^2} = 2 \\ \Leftrightarrow &p^2 = 2q^2 \end{aligned}$$

Also ist 2 ein Teiler von p^2. Es folgt, dass 2 auch ein Teiler von p ist, denn 2 ist prim, kann also nicht als Produkt anderer natürlicher Zahlen entstehen. Da nun 2 ein Teiler von p ist, können wir p schreiben als $p = 2 \cdot r$ mit $r \in \mathbb{N}$. Damit rechnen wir weiter:

$$\begin{aligned} &\frac{p}{q} \cdot \frac{p}{q} = 2 \\ \Leftrightarrow &\frac{2 \cdot r}{q} \cdot \frac{2 \cdot r}{q} = 2 \\ \Leftrightarrow &\frac{4r^2}{q^2} = 2 \\ \Leftrightarrow &4r^2 = 2q^2 \\ \Leftrightarrow &2r^2 = q^2 \end{aligned}$$

Also ist 2 ein Teiler von q^2 und somit auch ein Teiler von q. Dann ist aber $\mathsf{ggT}(p, q) \geq 2$, was im Widerspruch dazu steht, dass $\mathsf{ggT}(p, q) = 1$, also haben wir einen Widerspruch gefunden. Die einzige Annahme, die wir getroffen haben, ist, dass $\sqrt{2}$ rational ist, und somit muss das falsch sein und $\sqrt{2} \notin \mathbb{Q}$ ist bewiesen.

1.3.3 Diagonalisierung

Der Diagonalisierungsbeweis ist ein spezieller Widerspruchsbeweis, der mit Fragestellungen der Abzählbarkeit zu tun hat. Wir nennen eine Menge M abzählbar, wenn es eine surjektive Abbildung von $\mathbb{N}_0$ auf M gibt. Jede endliche Menge ist offensichtlich abzählbar und wir kennen mindestens eine unendliche Menge, die abzählbar ist, nämlich $\mathbb{N}_0$ vermöge der Identitätsfunktion. Dass es auch überabzählbare Mengen gibt, kann mithilfe eines Diagonalisierungsbeweises bewiesen werden. Ein Diagonalisierungsbeweis funktioniert wie folgt:

Definition 1.3.3 : Diagonalisierungsbeweis

Zu zeigen: Für die Menge M von Elementen, die eine Darstellung als Sequenz von Zeichen aus einem festen Alphabet Σ besitzen gibt es keine Aufzählung, das heißt keine surjektive Abbildung $f\colon \mathbb{N}_0 \to M$.
Vorgehen: Wir nehmen an, dass es eine solche surjektive Abbildung $f\colon \mathbb{N}_0 \to M$ gibt und konstruieren eine Tabelle T die in Spalte i und Zeile j das i-te Zeichen von $f(j)$ enthält. Wir betrachten dann das Element, das sich durch die Diagonale ausdrücken lässt, also das Element, das sich darstellen lässt als $T[1,1]T[2,2]T[3,3]$.... Wir modifizieren dieses Diagonalelement geeignet an jeder Stelle und erhalten so ein neues Element x. Zu zeigen ist nun, dass $x \in M$ gelten muss. Auf Grund der Konstruktion von x kann dann x aber an keiner Stelle in T auftauchen und somit kann M nicht abzählbar sein.

Das bekannteste Beispiel für einen Diagonalisierungsbeweis ist der Beweis, dass die reellen Zahlen $\mathbb{R}$ nicht abzählbar sind. Wir zeigen sogar, dass bereits das Intervall $[0,1[$ überabzählbar ist. Hierzu müssen wir nutzen (ohne, dass das an dieser Stelle bewiesen werde), dass jede reelle Zahl eine eindeutige Darstellung als (unendliche) Dezimalzahl besitzt und umgekehrt jede unendliche Dezimalzahl einer reellen Zahl entspricht. Wir nehmen jetzt an, dass es eine surjektive Funktion $f\colon \mathbb{N}_0 \to [0,1[$ gäbe. Dann konstruieren wir die Tabelle T derart, dass $T[i,j]$ die j-te Nachkommastelle von $f(i)$ enthält. Wir betrachten das Diagonalelement $T[1,1]T[2,2]T[3,3]$... und definieren x als $0,T'[1,1]T'[2,2]T'[3,3]$.... mit

$$T'[i,i] = \begin{cases} 1 & \text{falls } T[i,i] = 0 \\ 0 & \text{sonst} \end{cases}$$

Hierbei handelt es sich offensichtlich um eine Dezimaldarstellung und somit um eine reelle Zahl, also $x \in [0,1[$. Allerdings kann x an keiner Stelle in der Tabelle stehen. Denn angenommen, x wäre das i-te Element der Tabelle, dann müsste gelten $T[i,i] = T'[i,i]$, was nach Definition von T' offensichtlich nicht möglich ist. Also kann es eine surjektive Funktion $f\colon \mathbb{N}_0 \to [0,1[$ nicht geben und das Intervall $[0,1[$ ist somit nicht abzählbar. $\mathbb{R}$ als Obermenge von $[0,1[$ ist demnach auch nicht abzählbar.

Lektion 2: Klassen und Objekte

In Grundlagen der Informatik 1 haben Sie vorrangig einfache Datenstrukturen wie Zahlentypen, Zeichenketten oder bool-Variablen manipuliert. Neben diesen vordefinierten Datentypen gibt es in objektorientierten Sprachen wie C# die Objekttypen, die nicht von der Sprache vordefiniert sind, sondern durch Klassenvereinbarungen eingeführt werden. Dieses Kapitel stellt die Verwendung von Objekttypen vor. Insbesondere mit der Klasse String haben Sie bereits einen Objekttyp kennengelernt, aber bislang noch keine eigenen Objekttypen definiert. Die Definition von Objekttypen (Klassen genannt), deren Instanziierung und Manipulation sind die wesentlichen Merkmale, die die objektorientierte Programmierung von der imperativen Programmierung, die Sie bislang kennengelernt haben, abheben.

2.1 Objekttypen

Typen in C# werden grundsätzlich in Wert-Typen und Referenz-Typen unterteilt. Die meisten Typen, die Sie bislang verwendet haben, beispielsweise int, bool, double und struct sind so genannte Wert-Typen. Variablen von Wert-Typen enthalten den Wert selbst als Inhalt, wohingegen Variablen von Referenz-Typen nur einen Verweis auf den eigentlichen Wert enthalten. Ein wesentlicher Unterschied zwischen Wert- und Referenz-Typen ist, dass bei der Zuweisung einer Variable von einem Wert-Typ der Wert kopiert wird, wohingegen bei der Zuweisung eines Referenz-Typs nur ein Verweis gesetzt wird. Mit string und Feldern (Arrays) haben Sie bereits in der Vergangenheit Referenz-Typen kennen gelernt. Den Unterschied zwischen Referenz- und Wert-Typ illustriert das folgende Beispiel, wir definieren zunächst einen einfachen struct namens Pair:

```
struct Pair
{
    public int x;
    public int y;
}
```

und betrachten damit folgende Methode:

```
void refOrValueType()
{
    Pair p1 = new Pair();
    p1.x = 1;
    p1.y = 2;
```

```
    Pair p2 = p1;
    p2.y = 3;
    Console.WriteLine("p1 = (" + p1.x + ", " + p1.y + ")");
    Console.WriteLine("p2 = (" + p2.x + ", " + p2.y + ")");

    int[] a1 = new int[2];
    a1[0] = 1;
    a1[1] = 2;
    int[] a2 = a1;
    a2[1] = 3;
    Console.WriteLine("a1 = [" + a1[0] + ", " + a1[1] + "]");
    Console.WriteLine("a2 = [" + a2[0] + ", " + a2[1] + "]");
}
```

Die Ausgabe dieses Programms lautet:

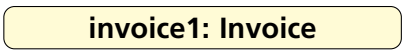
```
p1 = (1, 2)
p2 = (1, 3)
a1 = [1, 3]
a2 = [1, 3]
```

Man kann beobachten, dass p1 und p2 *verschiedene* Pair-Instanzen enthalten, wohingegen a1 und a2 auf dieselbe Entität verweisen. Eine Änderung an p2 hat also keine Auswirkungen auf p1, ändert man aber einen Eintrag in a2, so wird dabei auch das Feld, das a1 referenziert verändert – weil es sich um dasselbe handelt.

In dieser Lektion werden wir uns mit einem in C# besonders wichtigen Referenz-Typ befassen, nämlich mit Objekt-Typen. In Java ist es sogar so, dass Referenz-Typen und Objekt-Typen synonym sind. Felder (Arrays) und String sind in Java ebenfalls Objekt-Typen und die Wert-Typen wie int, double und boolean werden daher auch primitive Datentypen genannt.

2.1.1 Ein Objekt verwenden

Bei der objektorientierten Programmierung verkörpern Objekte reale oder abstrakte Gegenstände und Rollen der Anwendungswelt. Typischerweise existieren zur Laufzeit objektorientierter Programme mehrere Objekte, die durch den Austausch von Nachrichten miteinander interagieren. Wir beginnen unsere Betrachtung zunächst an einem einzelnen Objekt der Klasse Invoice die zur Verwaltung von Rechnungen gedacht ist, das einer Variablen mit dem Namen invoice1 zugewiesen ist:



Der Typ der Variablen invoice1 ist Invoice. Invoice ist ein Objekttyp. Anders als primitive Datentypen sind Objekttypen nicht (grundsätzlich) in C# vordefiniert. Ein Objekttyp Invoice existiert genau dann, wenn im Kontext seiner Verwendung eine Klasse Invoice definiert ist. Es gibt in der C# API einige vordefinierte Klassen, eine davon haben Sie bereits kennen gelernt: String. In dieser Lektion wollen wir uns aber auf Objekttypen konzentrieren, die Sie selbst definieren.

Bemerkung 2.1.1

Damit eine Klasse verwendet werden kann, muss sie am Ort der Verwendung sichtbar sein. Die Sichtbarkeit von Klassen, Methoden und Attributen[a] sowie Mechanismen zur Beeinflussung der Sichtbarkeit sind Gegenstand einer späteren Lektion. Solange keine ausdrücklichen Vorkehrungen zur Sichtbarkeit getroffen werden, umfasst der Kontext gewisse Klassen der C#-Laufzeitbibliothek (API) sowie diejenigen eigenen Klassen, die sich in demselben Projekt wie die aufrufende Klasse befinden.

[a] Wir verwenden in diesem Kurs den Begriff „Attribut" um Objekt- und Klassenvariablen zu bezeichnen, wie es in der Literatur zur objektorientierten Programmierung üblich ist. In C# gibt es ein Konzept „attribute", das im Grunde spezielle Klassen bezeichnet, die verwendet werden, um andere Klassen mit Tags zu versehen, die gewisse Meta-Informationen enthalten. Mit diesen „attributes" befassen wir uns in diesem Kurs nicht.

Rekapitulieren wir kurz, was wir aus der ersten Lektion – zunächst noch unabhängig von Sprachkonstrukten spezifischer Programmiersprachen – bereits über Objekte und Klassen wissen:

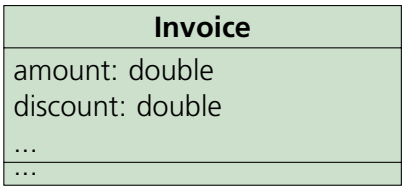
- Objekte sind Exemplare oder Instanzen[1] einer Klasse[2].
- Objekte haben Attribute, auch Instanz-, Objekt- oder Exemplarvariablen genannt.
- Attribute haben einen Typ und einen Attributwert.
- Der Zustand eines Objekts ist durch die Werte seiner Attribute festgelegt.
- Objekte haben Methoden, deren Ausführung durch das Senden einer Nachricht an das Objekt ausgelöst wird.
- Klassen beschreiben die Eigenschaften und das Verhalten aller ihrer Instanzen, denn durch die Klassenbeschreibung ist festgelegt, welche Attribute und Methoden Objekte dieser Klasse besitzen.

Um ein Objekt, beispielsweise das Objekt invoice1, korrekt verwenden zu können, benötigen wir einige Informationen über den Objekttyp. Um Attribute des Objekts invoice1 ansprechen zu können, benötigen wir Kenntnis über die Namen und Typen der Attribute der Klasse Invoice. Auch müssen wir die Methoden der Klasse Invoice kennen, um korrekte Nachrichten an das Objekt invoice1 senden zu können. Solche Informationen können auf unterschiedliche Art dokumentiert sein. Im Rahmen dieses Kapitels entnehmen wir sie aus UML-Klassendiagrammen, wie wir sie in der ersten Lektion kennengelernt haben.

[1] Instanzen ist ein Lehnwort aus dem Englischen „instance". Gemäß der ursprünglichen Bedeutung des Wortes „Instanz" im Deutschen ist die Begriffswahl unpassend, aber da der Begriff sich über Jahrzehnte etabliert hat und eine weitere Verbreitung hat als „Exemplar" werden wir den Begriff in diesem Skript verwenden.

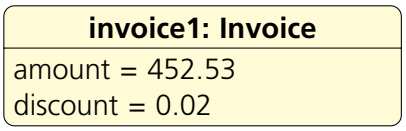
[2] Es gibt auch objektorientierte Sprachen ohne Klassenkonzept, aber in C# und verwandten Sprachen entstehen Objekte nur als Instanzen von Klassen. Das bekannteste Beispiel für eine prototypenbasierte objektorientierte Sprache, in der nicht jedes Objekt aus einer Klasse hervorgeht, ist JavaScript.

Wir betachten das folgende (vorläufige) Klassendiagramm für die Klasse Invoice, bei der wir drei Punkte verwenden um darzustellen, dass die Liste der Attribute und Methoden noch nicht vollständig ist:



Wir können dieser UML-Klassenbeschreibung entnehmen, dass jedes Objekt der Klasse Invoice über die double-Attribute amount und discount verfügt. Beachten Sie, dass nur die Typen, nicht jedoch die Werte der Attribute durch die Klassenbeschreibung festgelegt sind. Die Attributwerte können sich von Instanz zu Instanz unterscheiden und bilden den Zustand eines Objekts. Da die Befehle in C# und die API-Klassen englische Begriffe verwenden, nutzen wir in diesem Skript im Regelfall englische Namen für Klassen, Methoden und Attribute.

Das Objekt invoice1 könnte beispielsweise zu einem bestimmten Zeitpunkt folgende Attributwerte aufweisen:



In C# (und vielen anderen objektorientierten Programmiersprachen, auch in Java) lassen sich die Attribute des Objekts invoice1 wie folgt ansprechen:

```
invoice1.amount
invoice1.discount
```

Attributwerte lassen sich als Ausdrücke in Anweisungen auswerten:

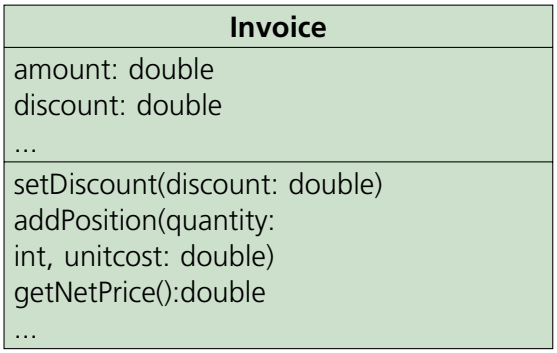
```
Console.WriteLine(invoice1.amount);
double netprice = invoice1.amount * (1-invoice1.discount);
```

Weiterhin kann Attributen mit dem Zuweisungsoperator = ein neuer Wert zugewiesen werden:

```
invoice1.amount = 12.34;
invoice1.discount = .3;
```

Bei der Auswertung wie bei der Zuweisung von Attributwerten gelten die Regeln der Typverträglichkeit, die Sie aus Grundlagen der Informatik 1 kennen. Das heißt, dass man einer Variable vom Typ T Werte vom Typ T und von jedem spezielleren Typ S zuweisen kann. Allgemeinere Typen oder Typen, die nicht mit T in einer Spezialisierungsrelation stehen, können hingegen nicht zugewiesen werden.

Wir erweitern jetzt unser vorläufiges UML-Diagramm für die Klasse Invoice um erste Methoden:



Wir erkennen, dass jede Methode einen Namen hat, der mit einem Klammerpaar endet. Über den Namen kann die Methode aufgerufen werden. Falls im Inneren des Klammerpaars ein oder mehrere Parameter angegeben sind, so bedeutet dies, dass die Methode bei ihrem Aufruf entsprechende Argumente benötigt. So muss beim Aufruf der Methode setDiscount ein double-Wert an das Objekt übergeben werden. Die Anweisung

```
invoice1.setDiscount(0.05)
```

bewirkt, dass die Methode setDiscount des Invoice-Objekts invoice1 mit dem Parameter 0.05 ausgeführt wird.

Bemerkung 2.1.2

Parameternamen, wie sie in der UML und auch in anderen Dokumentationsarten für Methoden oft angegeben werden, dienen dem Verständnis der Methode. Für den Methodenaufruf sind sie bedeutungslos. Das bedeutet in einem Kontext kann es nicht zwei verschiedene Methoden mit gleichem Namen und gleichen Parametertypen geben; f(int x) und f(int y) können also nicht im gleichen Bedeutungskontext existieren.

Die Methode addPosition benötigt als erstes Argument einen int-Wert für die Anzahl der Einheiten und als zweites Argument einen double-Wert für den Einzelpreis des neuen Rechnungspostens:

```
invoice1.addPosition(4, 12.67);
```

Bemerkung 2.1.3

Die Parameter müssen bei einem Methodenaufruf stets in der dokumentierten Reihenfolge übergeben werden. Eine Methode f(int x, String y) ist also von einer Methode f(String y, intx) zu unterscheiden. Beide können im gleichen Bedeutungskontext zusammen existieren.

Methoden können weiterhin ein Ergebnis zurück liefern, das außerhalb des Objekts weiter verwendet werden kann. Falls eine Methode ein Ergebnis zurück liefert, ist dies in einem

UML-Klassendiagramm am Rückgabetyp zu erkennen, der hinter dem Klammerpaar und einem Doppelpunkt angegeben ist. Die Methode getNetPrice liefert einen double-Wert an die aufrufende Stelle zurück, der beispielsweise in einer Zuweisung weiter verwendet werden kann:

```
double netprice = invoice1.getNetPrice();
```

Für die Übergabe von Argumenten wie für die Auswertung von Rückgabewerten gelten die Regeln der Typverträglichkeit.

Selbsttestaufgabe 2.1.1

Welche der folgenden Zeilen sind korrekte (ausführbare) Anweisungen?

```
1 invoice1.amount = 32.20 + 2.40;
2 invoice1.amount = 50;
3 invoice1.addPosition(3);
4 invoice1.setDiscount(0);
5 invoice1.getNetPrice(3 * 3.50);
```

Musterlösung zu Selbsttestaufgabe 2.1.1

Zeilen 1, 2 und 4 sind ausführbar. Zeile 3 hat einen Parameter zu wenig, Zeile 5 enthält Übergabeparameter, obwohl die Methode keine Parameter erwartet. Zudem wäre auch die syntaktisch korrekte Anweisung

```
invoice1.getNetPrice();
```

unnütz, solange das Ergebnis der Methodenausführung nicht weiter verwendet, z. B. einer Variablen zugewiesen wird.

2.1.2 Erzeugung und Lebensdauer von Objekten

Wenn man überlegen möchte, wie lange Daten im Speicher vorgehalten werden müssen, sind Referenz-Typen komplizierter als Wert-Typen. Wenn man einen Wert-Typen verwaltet, ist es leicht erkennbar, wie lang die Daten vorgehalten werden müssen, nämlich so lange, wie die zugehörige Variable Gültigkeit hat. Komplizierter ist das allerdings für Verweis-Typen und somit insbesondere Objekte. Ein Objekt muss explizit erzeugt werden, bevor es einer Variablen zugewiesen werden kann. Nach seiner Erzeugung existiert ein Objekt solange, wie ein Verweis auf das Objekt existiert. Ein solcher Verweis ist beispielsweise eine Variable, der das Objekt zugewiesen ist.

Ein Objekt, auf das kein Verweis mehr existiert, wird von C#s automatischer Speicherverwaltung (engl. garbage collection) aus dem Programmsystem entfernt. Genauso wird auch in Java verfahren, es gibt aber auch objektorientierte Programmiersprachen ohne

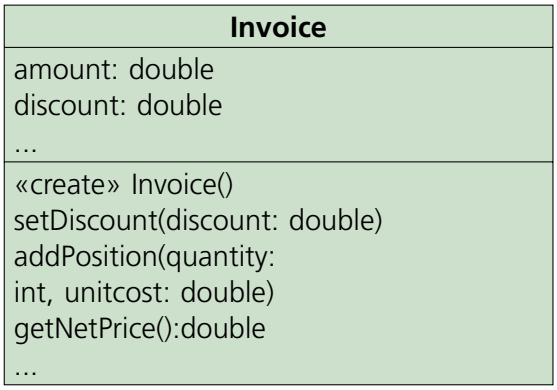
eine automatische Speicherverwaltung; beispielsweise C++ und Delphi erfordern eine manuelle Speicherverwaltung. Das heißt, wenn in einer Programmiersprache ohne automatische Speicherverwaltung ein Objekt erzeugt wird, ist es die Aufgabe des Programmierers, nachzuhalten, ob das Objekt noch existiert und es gegebenenfalls mit einem Destruktor zu zerstören, wenn es nicht mehr gebraucht wird. Das ist allerdings fehleranfällig, denn wenn man ein Objekt zerstört, von dem man fälschlicherweise annimmt, dass es nicht mehr benötigt wird, aber eine andere Variable, die auf das Objekt verweist, noch einmal angesprochen wird, dann kann es zu Fehlverhalten oder dem Auftreten einer Ausnahme kommen.

In C# müssen Sie sich um Fehler dieser Art üblicherweise[3] nicht kümmern, allerdings ist die automatische Speicherverwaltung mit einem gewissen Overhead verknüpft. Immer dann, wenn der Garbage Collector den Speicher aufräumt, kann das einen kurzfristigen Einfluss auf die Rechengeschwindigkeit der Anwendung haben. In Echzeitanwendungen, beispielsweise Videospielen, kann das gelegentlich ungünstig sein. Wenn man an einer Software arbeitet, in der phasenweise der Garbage Collector nicht angewendet werden sollte, kann es sinnvoll sein, bevor man eine solche Phase betritt, die Methode GC.Collect aufzurufen, um den Garbage Collector dazu zu zwingen, den Speicher zu genau diesem Zeitpunkt aufzuräumen. Denkbar ist es beispielsweise, dass beim Übergang zwischen zwei Levels in einem Videospiel – nach dem Verlassen des alten und vor dem Betreten des neuen Levels – ein solcher Aufruf sinnvoll sein kann. Für die allermeisten Anwendungen sollte man den Garbage Collector aber selbständig arbeiten lassen, um eine effiziente Speicherverwaltung zu erreichen.

Zur Erzeugung von Objekten einer bestimmten Klasse wird ein Konstruktor dieser Klasse aufgerufen. Ein Konstruktor ist eine spezielle Methode, die dazu dient, ein Objekt zu initialisieren, d. h. in einem definierten Anfangszustand in der Laufzeitumgebung bereitzustellen.

In C# und Java lautet der Name eines Konstruktors stets gleich wie der Klassenname und gibt, wenn man ihn aufruft, das durch ihn neu erzeugte Objekt zurück. In der UML können Konstruktoren zudem (optional) durch das Schlüsselwort << create >> gekennzeichnet werden. Das Klammerpaar eines Konstruktors kann leer sein oder eine Parameterliste enthalten. Eine Klasse kann mehrere Konstruktoren anbieten, die sich über ihre Parameterliste unterscheiden. Gibt man für eine Klasse keinen Konstruktor explizit an, so gibt es automatisch einen impliziten parameterlosten Standardkonstruktor – soweit das nicht durch Vererbung unmöglich ist, mehr dazu in der nachfolgenden Lektion 3. Wir erweitern in der folgenden Darstellung unser Klassendiagramm für Invoice um einen parameterlosen Konstruktor.

[3] Es gibt Fälle, in denen man mit nicht verwalteten Daten zu tun hat, beispielsweise wenn man eine C++-Bibliothek nutzt. In solchen Fällen ist für diese Daten eine manuelle Speicherverwaltung notwendig.



In C# und Java wird zur Erzeugung eines Objekts dem Konstruktor das Schlüsselwort new vorangestellt. Ein Objekt vom Typ Invoice können wir mit dem Ausdruck new Invoice() erzeugen. Die Erzeugung wird in der Regel mit einer Zuweisung verbunden. Durch die Anweisung

```
Invoice invoice2 = new Invoice();
```

wird eine Variable vom Typ Invoice deklariert und dieser ein neu erzeugtes Invoice-Objekt zugewiesen. Die Zuweisung kann auch an eine bereits zuvor deklarierte Variable erfolgen:

```
Invoice invoice3;
...
invoice3 = new Invoice();
```

Selbsttestaufgabe 2.1.2

Laden Sie die Datei Invoice.cs aus dem Zusatzmaterial im Moodle herunter und fügen Sie eine Main-Methode hinzu. Verfassen Sie innerhalb der Main-Methode eine Anweisungssequenz, die eine Variable vom Typ Invoice mit einem Invoice-Objekt initialisiert, dem Invoice-Objekt einige Rechnungsposten hinzufügt, einen Rabatt festlegt und zuletzt den Nettopreis am Bildschirm ausgibt. Kompilieren Sie Ihr Programm und führen Sie es aus. Alternativ zum Download der Klasse können Sie auch den folgenden Code abtippen oder kopieren:

```
public class Invoice
{
    private double amount;
    private double discount;
    public Invoice()
    {
    }
    public int getSum()
    {
        return 5;
```

```
    }
    public void setDiscount(double discount)
    {
        this.discount = discount;
    }
    public void addPosition(int quantity, double unitcost)
    {
        this.amount += quantity * unitcost;
    }
    public double getNetPrice()
    {
        return amount * (1 - discount);
    }
}
```

Musterlösung zu Selbsttestaufgabe 2.1.2

```
invoice.addPosition(3, 2.5);
invoice.addPosition(7, 3.25);
invoice.addPosition(2, 50.5);
invoice.setDiscount(.1);
Console.WriteLine(invoice.getNetPrice());
```

Wie wir gesehen haben, werden Variablen für Objekttypen syntaktisch ebenso vereinbart wie Variablen für die primitiven Typen, die in C# vordefiniert sind. Es bestehen jedoch zwei bedeutsame Unterschiede zwischen Variablen für Objekttypen und Variablen für primitive Typen.

Zunächst kann eine Variable eines Objekttyps nur dann deklariert werden, wenn dieser Objekttyp im Kontext der Deklaration definiert ist. Diese Einschränkung gilt verständlicherweise auch für die Erzeugung von Objekten. Demgegenüber sind einige vordefinierte Typen wie `int` oder `string` in C# global verfügbar, so dass Variablen – wie auch Literale – dieser Datentypen an jeder Stelle eines C#-Programms auftreten können.

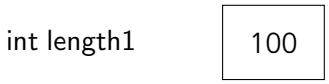
2.1.3 Wert-Variablen und Referenz-Variablen

Bemerkung 2.1.4

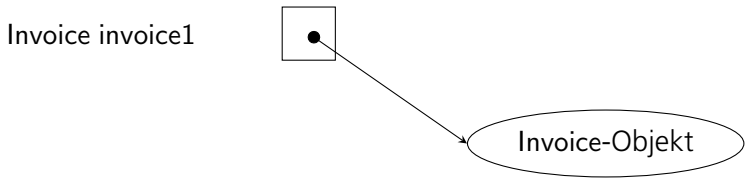
In diesem Kapitel gehen wir in den Beispielen davon aus, dass das Attribut `amount` in `Invoice` als `public` deklariert ist, auch wenn das kein guter Stil ist, um einfachere Beispiele angeben zu können.

Wir haben bereits über die Begriffe Wert-Typ und Referenz-Typ gesprochen. Diese Begriffe finden auch in der Klassifizierung von Variablen eine Entsprechung. Wir sprechen von Wert-Variablen und Referenz-Variablen (oder Verweis-Variablen).

Eine Variable ist eine benannte Speicherstelle im Arbeitsspeicher eines Rechners. Im Falle von Wert-Typen befindet sich an dieser Speicherstelle jeweils der Wert, der einer Variablen zugewiesen ist. Die folgende Abbildung zeigt schematisch die int-Variable length1 mit dem Wert 100.



Im Falle von Objekttypen befindet sich an der zugehörigen Speicherstelle nun aber nicht, wie man vielleicht vermuten könnte, das Objekt, das einer Variablen zugewiesen ist, sondern lediglich eine Referenz auf einen anderen Speicherbereich, der das eigentliche Objekt beherbergt. Die folgende Abbildung zeigt schematisch die Variable invoice1 vom Typ Invoice. Das ihr zugewiesene Invoice-Objekt befindet sich in einem anderen, von der Laufzeitumgebung verwalteten Speicherbereich, auf dessen Adresse die Speicherstelle von invoice1 verweist.



Zur Unterscheidung dieser beiden Arten von Variablen sprechen wir von Wert-Variablen und Referenz-Variablen. Wert-Variablen enthalten Wert-Typen und Referenz-Variablen enthalten Referenz-Typen. Alle Variablen für Objekttypen sind also Referenz-Variablen.

Bemerkung 2.1.5 : ref-Schlüsselwort und Wert-Variablen

In C#, nicht aber in Java, gibt es das Schlüsselwort ref, das für einen Methoden-Paramter angegeben werden kann. Ein Beispiel für eine Methode, die das Schlüsselwort ref verwendet wäre die folgende Methode:

```
int maximum(int[] array, ref int index)
{
    index = 0;
    for(int i = 1; i < array.Length; ++i)
        if (array[i] > array[index])
            index = i;
    return array[index];
}
```

Wenn man eine solche Methode aufrufen möchte, kann man für die Parameter, die mit ref gekennzeichnet sind, keine Literale, sondern nur Variablen angeben. Weiterhin muss man als Aufrufer ebenfalls den jeweiligen Paramter als ref kennzeichnen. Diese

Verpflichtung erhöht die Lesbarkeit des Codes, denn der Effekt eines solchen ref-Parameters ist, dass nicht etwa der Wert der Variable übergeben wird, sondern die Variable selbst. Wenn dann eine Änderung des Wertes in der aufgerufenen Methode stattfindet, dann ändert sich, quasi als Seiteneffekt, auch der Inhalt der übergebenen Variable am Aufrufpunkt. Ein Aufruf der Methode (durch eine andere Methode der gleichen Klasse) könnte so aussehen:

```
int index = 0;
int[] array = new int[] { 2, 4, 2, 5, 7, 3, 5, 1 };
int max = maximum(array, ref index);
Console.Write("Max an Position " + index + " ist " + max);
```

Man beachte die Angabe des Schlüsselwortes ref bei der Übergabe des zweiten Parameters. Die Ausgabe des Programms wäre:

```
Max an Position 4 ist 7
```

Es ist aber zu beachten, dass hierdurch nicht etwa – wie man meinen könnte – eine Referenz-Variable für einen Wert-Typen geschaffen wird, sondern dass schlicht in der aufrufenden Methode auf *dieselbe* Variable zugegriffen wird, die auch an der Aufruferstelle vorliegt. Es handelt sich also nur um ein lokales Alias.

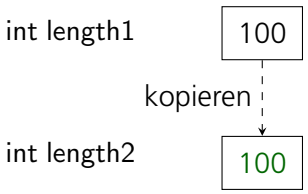
Der entscheidende Unterschied zwischen Wertvariablen und Verweisvariablen tritt bei Auswertungen, beispielsweise im Zusammenhang mit Zuweisungen der Art

```
var1 = var2;
```

zu Tage. Eine solche Zuweisung bewirkt, dass der Inhalt der Speicherstelle von var2 in die Speicherstelle von var1 kopiert wird. Handelt es sich bei var1 und var2 um Wert-Variablen, so wird der Wert von var2 in die Speicherstelle von var1 kopiert; der Effekt der Anweisungsfolge

```
int length1 = 100;
int length2 = length1;
```

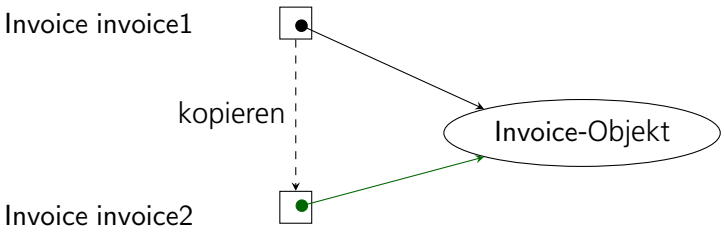
kann also wie folgt illustriert werden.



Nach der Anweisungsfolge sind sowohl length1 als auch length2 mit dem Wert 100 belegt. Wie wir wissen, sind diese Werte zwar gleich, aber voneinander vollkommen unabhängig. So bewirkt die nachfolgende Anweisung

```
length2 += 300;
```

dass length2 den Wert 400 erhält, wohingegen length1 weiterhin mit dem Wert 100 belegt ist. Handelt es sich bei var1 und var2 jedoch um Variablen eines Objekttyps (und somit notwendigerweise um Referenz-Variablen), so wird die Referenz auf ein Objekt in die Speicherstelle von var1 kopiert:



Das Resultat des Kopiervorgangs sind somit zwei identische Referenzen, so dass invoice1 und invoice2 auf ein und dasselbe Invoice-Objekt verweisen.

Während die Wertvariablen length1 und length2 nach dem Kopiervorgang voneinander unabhängig geblieben sind, sind die Verweisvariablen durch den Kopiervorgang miteinander gekoppelt worden. Nach Ausführung der Anweisungsfolge

```
Invoice invoice1 = new Invoice();
invoice1.amount = 100;
Invoice invoice2 = invoice1;
invoice2.amount += 300;
```

Ist sowohl invoice1.amount als auch invoice2.amount mit dem Betrag 400.0 belegt, denn es handelt sich bei invoice1.amount und invoice2.amount um ein und dasselbe Attribut eines einzigen Objekts.

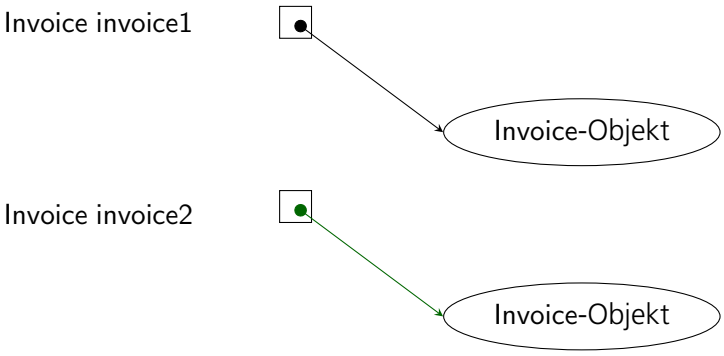
Betrachten wir die Vorgänge Schritt für Schritt:

- new Invoice veranlasst die Erzeugung eines Invoice-Objekts in einem von der Laufzeitumgebung verwalteten Speicherbereich und liefert eine Referenz auf das Objekt. Eine Invoice-Variable invoice1 wird deklariert und erhält durch Zuweisung diese Referenz.
- Dem amount-Attribut des von invoice1 referenzierten Invoice-Objekts wird der Wert 100.0 zugewiesen.
- Eine weitere Invoice-Variable invoice2 wird deklariert und erhält durch Zuweisung eine Kopie der in invoice1 gespeicherten Referenz. Die Zuweisung hat zur Folge, dass beide Variablen invoice1 und invoice2 auf dasselbe Invoice-Objekt verweisen.
- Der Wert des amount-Attributs dieses nun von invoice1 und invoice2 referenzierten Invoice-Objekts wird um 300.0 erhöht.

Die Koppelung zwischen zwei Verweisvariablen, die dasselbe Objekt referenzieren, wird bei einer Zuweisung an eine der beiden Variablen wieder aufgehoben. Nach Ausführung der Anweisung

```
invoice2 = new Invoice();
```

verweisen invoice1 und invoice2 auf zwei separate Invoice-Objekte mit jeweils eigenen, unabhängigen Zuständen:



Ob zwei Variablen von einem Objekt-Typen dasselbe Objekt oder zwei separate Objekte referenzieren, kann mit den Operatoren == bzw. != überprüft werden. Der Ausdruck

```
invoice1 == invoice 2
```

liefert dann und nur dann true, wenn Objekt-Identität vorliegt, d. h. wenn beide Variablen dasselbe Objekt referenzieren. Umgekehrt liefert der Ausdruck

```
invoice1 != invoice 2
```

genau dann true, wenn die beiden Variablen zwei separate Objekte referenzieren. Beachten Sie, dass zwei separate Objekte, deren Zustand (Attributwerte) gleich ist, nicht identisch sind. Nach der Ausführung von

```
Invoice invoice3 = new Invoice();
Invoice invoice4 = new Invoice();
bool isIdentical = (invoice3 == invoice4);
```

hat isIdentical den Wert false. Zwar wurden beide Invoice-Objekte mit gleichen (durch die Klasse Invoice definierten) Anfangszuständen erzeugt und nicht weiter verändert, doch verweisen die beiden Variablen invoice3 und invoice4 auf zwei separate Objekte; ihre Referenzen sind nicht identisch.

Bemerkung 2.1.6

Die Semantik des ==-Operators ist standardmäßig für alle Klassen als Identität der Referenz gegeben. Allerdings erlaubt C# im Gegensatz zu beispielsweise Java, diesen Operator zu überladen und somit für den Vergleich zweier Objekte der gleichen Klasse eine andere Semantik zu nutzen. Im Fall eines überladenen ==-Operators gelten die gerade angestellten Überlegungen nicht grundsätzlich. Darüber hinaus ist es außerdem so, dass ein Vergleich anderer Verweis-Variablen möglicherweise gar nicht erlaubt ist (weil der ==-Operator für diese nicht definiert wurde) oder eine abweichende Semantik von der Identität hat. Für Zeichenketten (String) beispielsweise wird mit == darauf überprüft, ob die Zeichenketten die gleichen Zeichenfolgen enthalten

– optional sogar mit lokalen Spezialitäten wie z. B. nicht-Unterscheidung von „ß“ und „ss“.

Eine Referenz-Variable (und damit auch jede Variable von einem Objekt-Typen) kann auch den undefinierten Verweis null enthalten. Man spricht auch von einer null-Referenz. Das Schlüsselwort null können wir in Verbindung mit einem Gleichheits- oder Ungleichheitsoperator sowie in Zuweisungen verwenden. Der Ausdruck

```
invoice5 == null
```

liefert true, genau dann wenn invoice5 auf kein Objekt verweist. Der Ausdruck

```
invoice5 != null
```

liefert in diesem Fall false. Wird ein referenziertes Objekt nicht mehr benötigt, so kann es durch

```
invoice5 = null
```

von der referenzierenden Variablen gelöst werden. Falls es keine anderen Verweise auf dasselbe Objekt gibt, wird es in der Folge durch die automatische Speicherverwaltung entfernt.

Die Verwendung von null kann fehleranfällig sein, denn wenn eine Verweisvariable den Wert null trägt, dann kann man natürlich mit dieser Variable nicht auf Attribute oder Methoden zugreifen, die dem Typ nach vorhanden sein sollten. Beispielsweise kann der folgende Quellcodeschnipsel:

```
Invoice invoice6 = null;
invoice6.getSum();
```

kompiliert werden, aber wenn man ihn ausführt, unterbricht die Programmausführung und es wird eine NullReferenceException ausgegeben. Das passiert immer dann, wenn man auf einer null-Referenz Operationen ausführen möchte. Diese Fehler können bisweilen schwierig zu finden sein, da sie nicht zur Compile-Zeit auftreten, sondern erst zur Ausführungszeit. Daher bietet C# optional die Möglichkeit, zu fordern, dass Referenzvariablen nicht den Wert null annehmen sollen. Hierzu können Sie das Pragma #nullable enable in Ihrer Quelltextdatei verwenden. Dann geht der Compiler davon aus, dass eine Referenzvariable niemals null sein darf, außer bei der Deklaration folgt auf den Typ ein Fragezeichen:

```
1 #nullable enable
2 Invoice invoice7 = null;
3 Invoice? invoivce8 = null;
```

Dieser Code kompiliert zwar und kann auch ausgeführt werden, aber der Compiler gibt die folgende Warnung für Zeile 2 aus:

> Das NULL-Literal oder ein möglicher NULL-Wert wird in einen Non-Nullable-Typ konvertiert.

In Zeile 3 hingegen wird ausdrücklich angegeben, dass invoice8 auch null sein könnte, so dass diese Zuweisung keine Warnung hervorruft.

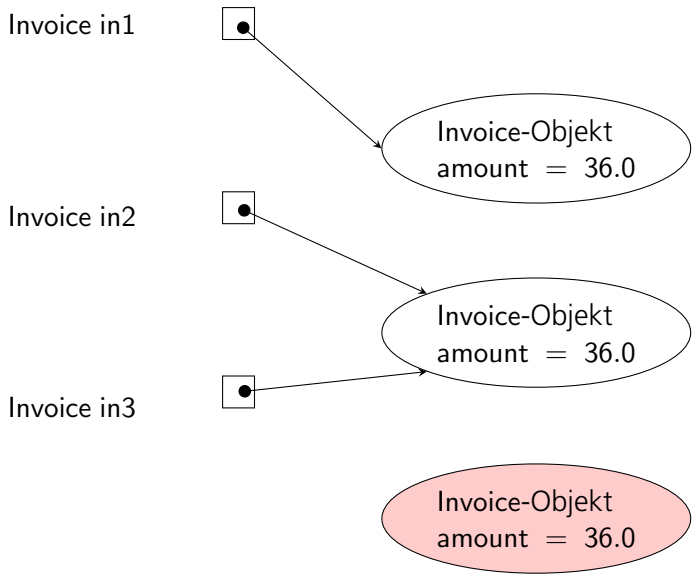
Auf diese Weise können Sie den Compiler nutzen, um Stellen, an denen potenziell null-Zuweisungen erfolgen können, zu identifizieren und so mögliche Fehler zu beheben.

Selbsttestaufgabe 2.1.3

Stellen Sie dar, welche Variablen nach der Ausführung der folgenden Anweisungen auf welche Objekte verweisen. Welche Werte besitzen die amount-Attribute der Objekte?

```
Invoice in1 = new Invoice();
in1.amount = 12;
Invoice in2 = new Invoice();
in2.amount = 24;
Invoice in3 = new Invoice();
in3.amount = 36;
in2.amount = in3.amount;
in3 = in2;
in1.amount = in3.amount;
```

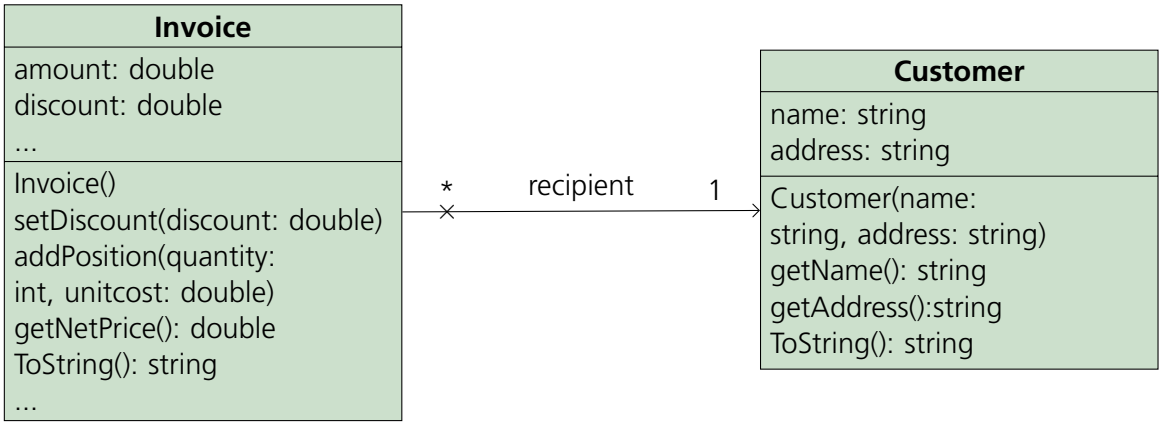
Musterlösung zu Selbsttestaufgabe 2.1.3



Wird von der Speicherverwaltung gelöscht

2.1.4 Zusammenspiel von Objekten

Wir beenden dieses Kapitel mit einem Anwendungsbeispiel, das in das Zusammenspiel von Objekten einführt. Das folgende UML-Klassendiagramm zeigt noch einmal die Klasse Invoice, weiterhin eine neue Klasse Customer und eine Assoziation zwischen diesen beiden Klassen.



Betrachten wir zunächst die neue Klasse Customer. Sie verfügt über zwei Attribute des Typs string. Der Typ string wird auch in den Parametern des Konstruktors erwartet und tritt in den Rückgabewerten der beiden Methoden auf.

Wir können einen Customer beispielsweise mit dem folgenden Befehl erzeugen:

```
Customer c = new Customer(
    "Anna Arndt", "Akazienallee 1, 52062 Aachen");
```

Um die Kundendaten abzufragen, sind Objekte der Klasse Customer mit zwei Methoden getName und getAddress ausgestattet. Die Anweisung

```
string customerName = c.getName();
```

liefert die Zeichenkette "Anna Arndt" an eine neue string-Variable customerName, die außerhalb des Customer-Objekts weiter verwendet werden kann.

Die Assoziation zwischen den Klassen Customer und Invoice im Klassendiagramm lässt uns erkennen, dass Invoice-Objekte jeweils genau ein Customer-Objekt kennen, während Customer-Objekte bei beliebig vielen Invoice-Objekten bekannt sein können, aber diese selbst nicht kennen. Die Umsetzung dieses Sachverhalts in C# geschieht dadurch, dass die Klasse Invoice über ein Attribut recipient vom Typ Customer verfügt. Erstmalig treffen wir hier auf den Sachverhalt, dass eine Klasse über ein Attribut einer anderen Klasse verfügt. Solche Konstrukte sind charakteristisch für die objektorientierte Programmierung.

Bemerkung 2.1.7

Attribute primitiver Datentypen werden in der Attributliste einer Klasse aufgeführt. Ein Attribut eines Objekttyps kann entweder in der Attributliste aufgeführt oder durch

eine Assoziation kenntlich gemacht werden. Die Darstellung durch eine Assoziation empfiehlt sich insbesondere bei der gleichzeitigen Darstellung des zugehörigen Objekttyps. Durch Pfeilspitzen kann kenntlich gemacht werden, in welchem Objekttyp das Attribut angelegt werden soll. Wenn eine Assoziation ohne Pfeilspitze verwendet wird, ist die Umsetzung der Assoziation zwischen den involvierten Objekttypen offen gelassen.

Wir erzeugen ein neues Invoice-Objekt und weisen seinem Attribut recipient vom Typ Customer das Objekt c zu:

```
Invoice inv = new Invoice();
inv.recipient = c;
```

Erinnern wir uns an dieser Stelle, dass in einer Variablen eines Objekttyps ein Verweis auf ein Objekt (oder die null-Referenz) gespeichert ist. Sollte das Invoice-Objekt inv eine Änderung am Customer-Objekt seines Attributs recipient vornehmen, so wäre unmittelbar das von c referenzierte Objekt verändert. Der umgekehrte Fall ist ebenso möglich, wie wir abschließend kurz demonstrieren.

Wir wollen unserem Objekt inv noch einen Rechnungsposten hinzufügen, einen Rabatt festlegen und anschließend einen Ausdruck der Rechnung anfordern:

```
inv.addPosition(25, 1.76);
inv.setDiscount(0.04);
Console.WriteLine(inv.ToString());
```

Die Ausgabe könnte (je nach Implementation der ToString-Methoden) zum Beispiel so aussehen:

```
An: Anna Arndt
Akazienallee 1, 52062 Aachen
42,24
```

Nehmen wir nun an, die Anschrift der Kundin Anna Arndt hätte sich geändert. Wir passen das Objekt c entsprechend an und lassen anschließend erneut einen Rechnungsausdruck erstellen:

```
c.address = "Azurenallee 1, 73430 Aalen";
Console.WriteLine(inv.ToString());
```

Da wir wissen, dass das Objekt inv eine Referenz auf dasselbe Objekt c enthält, dessen Attribut address wir soeben manipuliert haben, verwundert uns der resultierende Bildschirmausdruck nicht:

```
An: Anna Arndt
Azurenallee 1, 73430 Aalen
42,24
```

Auf eine Gefahr im Umgang mit Referenz-Variablen wollen wir noch aufmerksam machen. Sei inv2 ein Invoice-Objekt, und wir wollen wissen, an welchen Kunden die Rechnung gerichtet ist. Wir verschaffen uns zunächst eine Referenz auf das Attribut recipient:

```
Customer c2 = inv2.recipient;
```

Sodann versuchen wir, mit der Anweisung

```
String name = c2.name;
```

den Namen des Kunden in Erfahrung zu bringen. Im ungünstigeren Fall scheitert die Auswertung von c2.name zur Laufzeit mit der Fehlermeldung

```
System.NullReferenceException
```

Der Fehler tritt auf, wenn das Attribut inv2.recipient und damit auch unsere Variable c2 eine null-Referenz enthält. Zwar kann eine null-Referenz problemlos an eine Customer-Variable zugewiesen werden, doch nur ein Customer-Objekt hätte ein string-Attribut name. Ein Methodenaufruf an einer null-Referenz würde zu demselben Problem führen.

Im Allgemeinen können wir nicht sicher wissen, ob sich hinter einer Verweisvariablen ein Objekt oder eine null-Referenz verbirgt. Es ist deshalb im Sinne einer robusten Programmierung ratsam, durch eine Konstruktion wie

```
string name;
if(c2 != null)
    name = c2.name;
else
    name = "kein Empfänger";
```

diese Unsicherheit abzufangen.

Selbsttestaufgabe 2.1.4

Erzeugen Sie zwei Customer-Objekte customer2 und customer3 mit beliebigen Namen und Anschriften. Schreiben Sie eine Anweisung, die dem Objekt customer2 die Anschrift von customer3 zuweist.

Musterlösung zu Selbsttestaufgabe 2.1.4

```
Customer customer2 = new Customer
    ("Bernd Bauer", "Bauernstraße 2, 28195 Bremen");
Customer customer3 = new Customer
    ("Carla Caesar", "Cardinalstraße 3, 27472 Cuxhaven");
customer2.address = customer3.address;
```

Selbsttestaufgabe 2.1.5

Können Sie einem Invoice-Objekt auch zwei Customer-Objekte zuordnen? Falls ja,

schreiben Sie eine passende Anweisungsfolge. Falls nein, begründen Sie.

Musterlösung zu Selbsttestaufgabe 2.1.5

Das ist nicht möglich, vgl. UML-Diagramm: einem Invoice-Objekt ist genau ein Customer-Objekt als recipient zugeordnet.

Selbsttestaufgabe 2.1.6

Können Sie einem Customer-Objekt auch zwei Invoice-Objekte zuweisen? Falls ja, schreiben Sie eine passende Anweisungsfolge. Falls nein, begründen Sie.

Musterlösung zu Selbsttestaufgabe 2.1.6

```
Invoice invoiceA = new Invoice();
Invoice invoiceB = new Invoice();
Customer customer4 = new Customer
    ("Dora Dorsch", "Dorfstraße 4, 44135 Dortmund");
invoiceA.recipient = customer4;
invoiceB.recipient = customer4;
```

Selbsttestaufgabe 2.1.7

Erzeugen Sie ein Invoice-Objekt, fügen Sie einige Rechnungsposten hinzu und legen Sie einen Rabatt fest. Weisen Sie kein Customer-Objekt zu. Was passiert, wenn nun die Methode ToString ausgeführt wird?

Musterlösung zu Selbsttestaufgabe 2.1.7

Das hängt von der konkreten Implementation ab. In der Referenzimplementation, die in diesem Kapitel genutzt wurde, wird eine NullReferenceExeption geworfen. Es wäre aber auch denkbar, eine robuste Implementation der ToString-Methode anzugeben, die einen Text wie „kein Rechnungsempfänger angegeben“ ausgibt.

2.2 Klassenelemente

Im letzten Kapitel haben wir bestehende Klassen genutzt. Im Laufe dieses Kapitels werden wir lernen, wie in C# Klassen deklariert werden und welche Bestandteile in einer solchen Deklaration enthalten sein können.

Klassen sind die wichtigsten Bausteine in C#-Programmen. Eine Klasse ist ein Gebilde, das dazu dient, Objekte mit der gleichen Menge von Attributen und Methoden zu definieren. Der Klassenname bezeichnet typischerweise ein Fachkonzept.

Eine Klasse ist ein Bauplan, der beschreibt, wie ein Objekt auszusehen hat, wenn es erzeugt wird. Das Objekt ist eine Instanz oder ein Exemplar (engl. instance) der Klasse. Wir können uns eine Klasse einerseits als eine Fabrik vorstellen, die Objekte erzeugt, und auf der anderen Seite als Schablone, die die Eigenschaften und das Verhalten der Objekte beschreibt.

2.2.1 Klassenvereinbarung

Klassen sind der hauptsächliche Strukturierungs- und Abstraktionsmechanismus in C#. Klassendeklarationen verkörpern das gesamte Wissen über Objekte, da alle Objekte Instanzen von Klassen sind.

Jede Klasse muss deklariert werden, damit sie benutzt werden kann. In diesem und in nachfolgenden Abschnitten lernen wir nach und nach die Anatomie von Klassenvereinbarungen im Detail kennen, um es Ihnen so zu ermöglichen eigene Klassen für gegebene Probleme zu erstellen. Die Bestandteile einer solchen Vereinbarung ähneln den Elementen einer Klasse in der UML.

Definition 2.2.1 : Klassenvereinbarung in C# (Grundform)

Die Klassenvereinbarung in C# hat die folgende Struktur:

```
class Klassenname
{
    Attributdeklarationen
    Konstruktordeklarationen
    Methodendeklarationen
}
```

Dabei sind alle drei inneren Bestandteile optional. Die Reihenfolge der inneren Bestandteile ist eine Empfehlung, grundsätzlich können Sie Attributdeklarationen, Konstruktordeklarationen und Methodendeklarationen in beliebiger Reihenfolge angeben und sogar mischen.

In den meisten Kontexten ist es, wenngleich in C# nicht zwingend, sinnvoll, die Deklarationen innerhalb der Klassendefinition im Block in der angegebenen Reihenfolge anzugeben. C# und Java erlauben es, davon abzuweichen und moderne Entwicklungsumgebungen helfen dabei, dennoch die Übersicht zu bewahren, es ist aber im Sinne der einfachen

Wartbarkeit des Codes äußerst empfehlenswert, sich an eine einheitliche Ordnung der Klassenelemente zu halten. Die hier vorgeschlagene Ordnung orientiert sich an der Ordnung in UML-Klassendiagrammen.

Es kann hilfreich sein, in größeren Klassen so genannte Regionen zu definieren, in denen man beispielsweise Attribute oder Methoden anlegt. Auch ist es manchmal sinnvoll, Methoden, die zu einer bestimmten Funktionalität gehören, zu einer Region zusammenzufassen. Diese Regionen kann man dann in Visual-Studio ein- und ausklappen und so die Übersichtlichkeit erhöhen. Eine Region definiert man mit dem Pragma #region und gibt ihr einen Namen. Möchte man beispielsweise die obigen Aspekte der Klassendefinition mit Regionen ausdrücken, sähe das wie folgt aus:

```
class Klassenname
{
    #region attributes
    Attributdeklarationen
    #endregion

    #region constructors
    Konstruktordeklarationen
    #endregion

    #region methods
    Methodendeklarationen
    #endregion
}
```

Eine Klassenvereinbarung (engl. class declaration) beginnt mit dem Schlüsselwort class. Eine Klassenvereinbarung legt einen neuen Objekttyp fest und spezifiziert seine Implementation im Rumpf der Klassenvereinbarung. Die Bestandteile des Rumpfs werden wir in den nächsten Abschnitten kennen lernen.

Bemerkung 2.2.2

Klassennamen sollten mit Großbuchstaben beginnen. Im Sinne der sprachlichen Kohärenz nutzen wir in diesem Skript üblicherweise englische Substantive um Klassen zu bezeichnen. Es ist sinnvoll, sich in dieser Hinsicht in jedem Projekt mit seinen Kollegen auf eine Konvention bei der Benennung der Klassen zu einigen.

2.2.2 Attributdeklaration

Wir erinnern uns, dass die Attribute (engl. attributes, fields) einer Klasse beschreiben, welche Eigenschaften die Objekte dieser Klasse aufweisen. Die konkreten Attributwerte (d. h. die konkreten Werte, die in der Attributvariable gespeichert sind) unterscheiden sich jedoch von Objekt zu Objekt. Wie schon in der UML müssen wir auch in C# die Attribute der Klasse deklarieren. Die Klasse Invoice besitzt beispielsweise die Attribute amount,

discount und vat. Wie Variablen haben auch Attribute einen Typ und einen Namen. Der Typ charakterisiert sowohl die Menge von Werten, die das Attribut annehmen kann, als auch die Menge von Operatoren oder Methoden, die auf diese Attribute anwendbar sind. Eine Attributdeklaration hat die gleiche Form wie eine Variablendeklaration, sie besteht folglich aus dem Typen und dem Namen des Attributs. Sie befindet sich direkt innerhalb des Klassenrumpfes. Attribute können sowohl Wert- als auch Referenz-Variablen sein. In unserem Beispiel sind alle Attribute vom Typ double.

```
public class Invoice
{
    public double amount = 0;
    public double discount;
    public double vat;
}
```

Bemerkung 2.2.3

Wir verwenden für Attribute üblicherweise englische Substantive und schreiben das erste Zeichen klein. Eine Attributdeklaration kann genauso wie eine Variablendeklaration eine Initialisierung beinhalten. Enthält eine Attributdeklaration keine Initialisierung und wird das Attribut vor seiner Benutzung nicht anderweitig initialisiert, so besitzt das Attribut einen Standardwert. Dieser ist bei allen numerischen Datentypen und Zeichen 0 bzw. 0.0, bei bool der Wert false und bei Variablen von einem Objekt-Typen null. Das Schlüsselwort public schreiben wir vor jedes Attribut, um auf die Attribute außerhalb der Klasse zugreifen zu können. Die genaue Bedeutung dieses Schlüsselwortes werden wir in der nächsten Lektion im Zuge der Sichtbarkeitskontrolle besprechen.

Alternativ nennen wir Attribute ohne weitere Schlüsselwörter wie static oder const auch Instanzvariablen oder Exemplarvariablen, weil jedes solche Attribut ein Mal je Instanz (oder Objekt) einer Klasse existiert. Auf Grund der Namensgleichheit mit Variablen von einem Objekt-Typen nutzen wir den Begriff Objektvariablen nicht, da das zu Verwechslungen führen kann.

Selbsttestaufgabe 2.2.1

Deklarieren Sie eine Klasse Circle mit dem Attribut radius um einen Kreis zu modellieren sowie die Klasse Rectangle mit den Attributen height und width um ein Rechteck zu modellieren.

Musterlösung zu Selbsttestaufgabe 2.2.1

```
class Circle
```

```
{
    public double radius;
}

class Rectangle
{
    public double width;
    public double height;
}
```

Selbsttestaufgabe 2.2.2

Deklarieren Sie die Klasse Customer mit den Attributen name und address um einen Kunden zu modellieren.

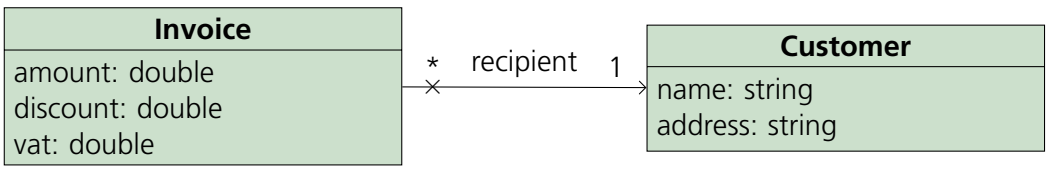
Musterlösung zu Selbsttestaufgabe 2.2.2

```
class Customer
{
    public string name;
    public string address;
}
```

Wie wir schon im vorangegangenen Kapitel gelernt haben, können Assoziationen in C# in Form von Instanzvariablen umgesetzt werden.

Selbsttestaufgabe 2.2.3

Versuchen Sie, die gerichtete Assoziation zwischen den Klassen Invoice und Customer in C# zu implementieren.



Musterlösung zu Selbsttestaufgabe 2.2.3

```
class Invoice
```

```
{
    public Customer recipient;
    public double amount;
    public double discount;
    public double vat;
}
```

Im Unterschied zu Attributen, die die Eigenschaften beschreiben, implementieren Methoden das Verhalten, das für alle Objekte dieser Klasse gültig ist. Die Deklaration von Methoden wird im folgenden Abschnitt behandelt. Idealerweise sollten Objekte von außen nur über Methodenaufrufe manipuliert werden.

2.2.3 Methodendeklaration

Nachdem wir uns zuvor mit der Deklaration von Attributen beschäftigt haben, wenden wir uns jetzt der Deklaration von Methoden (engl. method) und der Verhaltensbeschreibung im Methodenrumpf zu.

Eine Methode implementiert das gemeinsame Verhalten für alle Objekte einer Klasse. Methoden können den Zustand eines Objekts ändern, Informationen über das Objekt preisgeben oder neue Informationen erzeugen. Eine Methodendeklaration enthält im Allgemeinen ausführbaren Quelltext, der aus dem Kontext, in dem die Methodendeklaration gültig ist, aufgerufen werden kann.

Jede Methode besitzt einen Namen. Darüber hinaus kann sie einen Ergebnistyp festlegen, falls sie ein Ergebnis liefert, wie zum Beispiel eine Methode, die die Mehrwertsteuer der Rechnung berechnet. Eine Methode, die nur die Informationen zur Rechnung am Bildschirm ausgibt, besitzt keinen Ergebnistyp. Formal bezeichnen wir in diesem Fall den Rückgabetypen mit dem Schlüsselwort *void*. Als Ergebnistyp sind sowohl primitive Datentypen als auch Objekttypen zulässig. Des Weiteren kann eine Methode noch Parameter festlegen, die ihr bei einem Aufruf in Form von passenden Argumenten übergeben werden müssen. Wie beim Rückgabetyp gilt, dass Parameter sowohl einen primitiven Datentyp als auch einen Objekttyp haben dürfen.

Wenn wir von den Parametern einer Methode sprechen, kann es hilfreich sein, zwischen den formalen und den tatsächlichen (manchmal fehlübersetzt als „aktuellen“, aus dem Englischen „actual“) Parametern zu unterscheiden. Der formale Parameter bezeichnet den Parameter, wie er in der Parameterliste angegeben ist, wohingegen der tatsächliche Parameter den bei einem Methodenaufruf *tatsächlich* übergebenen Wert bezeichnet. Der tatsächliche Paramter unterscheidet sich von Methodenaufruf zu Methodenaufruf, wohingegen der formale Parameter über alle Methodenaufrufe der selben Methode hinweg identisch bleibt.

Definition 2.2.4 : Methodendeklaration

Eine Methode hat einen Kopf und einen Rumpf. Die grundlegende Methodendeklaration hat folgende Form:

```
Rückgabetyp Methodenname(Parameterliste)
{
    Anweisungen
}
```

Der Kopf in der ersten Zeile:

- zeigt an, ob die Methode ein Ergebnis liefert und, wenn ja, von welchem Typ das Ergebnis ist. Falls die Methode kein Ergebnis liefert, ist der Ergebnistyp void. Anderenfalls ist der Ergebnistyp ein in dem Kontext der Methode sichtbarer Typname.
- führt einen neuen Namen Methodenname ein und
- umfasst eine – möglicherweise leere – Liste formaler Parameter, die Parameterliste.

Eine Liste formaler Parameter hat die folgende Form:

```
t1 n1, t2 n2, ..., tm nm
```

wobei n1, n2, ..., nm paarweise verschiedene Namen der formalen Parameter sind und t1, t2, ..., tm deren Typen.

Der Methodenrumpf umfasst eine Folge von Anweisungen, wie wir sie in Grundlagen der Informatik 1 kennengelernt haben.

So können wir die Klasse Invoice zum Beispiel um die Methode setDiscount ergänzen. Da die Methode kein Ergebnis erzeugt, ist sie als void deklariert. Sie benötigt einen double-Wert – den neuen Rabatt – als Parameter.

```
class Invoice
{
    double amount = 0;
    double vat;
    double discount;
    Customer recipient;
    public void setDiscount(double newDiscount)
    {
        discount = newDiscount;
    }
}
```

Selbsttestaufgabe 2.2.4

Ergänzen Sie die Deklaration der Klasse Invoice um die Methode setVat, die die neue Mehrwertsteuer als Parameter besitzt, sowie die Methode addPosition, die die Stückzahl, sowie den Stückpreis übergeben bekommt und den Rechnungsbetrag um den Wert des Rechnungspostens erhöht. Die Methodenrümpfe können Sie zunächst leer lassen.

Musterlösung zu Selbsttestaufgabe 2.2.4

```
class Invoice
{
    double amount = 0;
    double vat;
    double discount;
    Customer recipient;
    public void setVat(double newVat){}
    public void addPosition(int quantity, double unitcost){}
}
```

Definition 2.2.5 : Signatur einer Methode

Die Signatur einer Methode besteht aus dem Namen der Methode und den Typen (sowie der Reihenfolge) der formalen Parameter.

Bemerkung 2.2.6

Achtung: Die hier verwendete Definition von Signatur ist nicht die einzige, die in C# verwendet wird. Je nach Kontext werden noch weitere Eigenschaften einer Methode zur Signatur gezählt. Die offizielle Microsoft-Dokumentation listet beispielsweise – für den Kontext von Delegaten, die wir in diesem Kurs aber nicht besprechen werden – zusätzlich den Rückgabetypen, die Sichtbarkeitsklasse (bisher haben wie hier nur public verwendet; wir werden die Sichtbarkeitsklassen noch ausführlich besprechen) und optionale Modifikatoren wie abstract oder sealed als Bestandteil der Signatur. Im Kontext dieses Kurses verwenden wir immer die oben angegebene Definition, aber beachten Sie, dass Sie in anderen Texten zu objektorientierter Programmierung oder C# im Speziellen auf abweichende Definitionen stoßen können.

Selbsttestaufgabe 2.2.5

Bestimmen Sie die Signaturen der Methoden der Klasse Circle, die hier ausschnittweise angegeben ist:

```
class Circle
{
    #region methods
    public double computeArea()
    {
        ...
    }
    public double computeCircumference()
    {
        ...
    }
    #endregion
}
```

Musterlösung zu Selbsttestaufgabe 2.2.5

Die Signaturen lauten:

- computeArea()
- computeCircumference()

Selbsttestaufgabe 2.2.6

Welche der nachfolgenden Methoden besitzen die gleiche Signatur?

```
public void a(int x)

public int a(double x)

public int a(int u)

public int b(double x, int y)

public void b(int x, double y)

public void c(int a, int b)
```

```
public void c(int a, int b, int x)

public void d(int x)
```

Musterlösung zu Selbsttestaufgabe 2.2.6

Da der Name der Methode zur Signatur gehört, können nur Paare von Funktionen mit dem gleichen Namen die gleiche Signatur haben. Damit kann d nicht die gleiche Signatur wie eine andere Methode in der Liste haben. Die beiden c-Methoden unterscheiden sich in der Anzahl ihrer formalen Parameter und die beiden b-Methoden in der Reihenfolge der Parametertypen. Die zweite a-Methode unterscheidet sich von der ersten und dritten im Typ ihres Parameters. Die erste und dritte a-Methode hingegen stimmen im Namen und den Typen in der Parameterliste überein, also besitzen sie dieselbe Signatur.

Grundsätzlich ist man in der Benennung seiner Methoden frei, solange man nur zugelassene Zeichen für die Benennung verwendet. Allerdings kann die Wahl des Namens für eine neu zu definierende Methode durch den Kontext in dem sie steht, eingeschränkt sein. In einer Klasse darf es niemals mehrere Methoden mit der gleichen Signatur geben, sonst führt dies zu einem Übersetzungsfehler. Da die Signatur sich aber aus dem Methodennamen und den Parametertypen zusammensetzt, ist es möglich, dass innerhalb einer Klasse mehrere Methoden mit dem gleichen Namen, aber mit verschiedenen Parametertypen existieren. Man spricht in diesem Fall von einem *überladenen* Methodennamen. Bei einem Aufruf mit diesem Methodennamen wird an Hand der Typen der Argumente entschieden, welche der Methoden ausgeführt wird.

Grundsätzlich müssen überladene Methodennamen keine gemeinsame Semantik bedeuten. Es kann also eine Methode mit der Signatur f(int x) und eine Methode mit der Signatur f(string x, int y) geben, die einen völlig unterschiedlichen Zweck erfüllen. Das sollte man aber unbedingt vermeiden, um die Lesbarkeit und Verwendbarkeit der selbst definierten Klassen und Methoden sicherzustellen. Ein Beispiel für ein sinnvolles Überladen einer Methode ist das folgende Paar von Methoden max:

```
int max(int x, int y)
{
    if(x>y) return x;
    return y;
}
int max(int[] x)
{
    int result = Int32.MinValue;
                int index = -1;
```

```
    while(++index < x.Length)
        result = max(result, x[index]);
    return result;
}
```

Selbsttestaufgabe 2.2.7

Wir betrachten die folgende Klasse (ohne Angabe der Methodenrümpfe) X, benennen Sie alle überladenen Methoden.

```
class X
{
    public void a(int x)

    public int a(double x)

    public int b(double x, int y)

    public void b(int x, double y)

    public void c(int a, int b)

    public void c(int a, int b, int x)

    public void d(int x)
}
```

Musterlösung zu Selbsttestaufgabe 2.2.7

Alle Methodennamen außer d sind überladen, also a, b und c.

Betrachten wir nun die Methode ToString, die neben dem Namen und der Anschrift des Rechnungsempfängers noch den Nettopreis inklusive Rabatt, die Mehrwertsteuer und den Bruttopreis der Rechnung als Zeichenkette zurückgeben soll. Sie muss dafür unter anderem auf die Werte der eigenen Attribute zugreifen. Ein solcher Zugriff erfolgt genauso wie ein Zugriff auf eine Variable. Der Zugriff auf Attribute des Customer-Objekts erfolgt mithilfe des .-Operators wie in Kapitel 2.1.4 gezeigt. Zusätzlich benötigen wir in der Methode Hilfsvariablen, um zum Beispiel den Netto- und den Bruttopreis zu berechnen. Diese Variablen nennen wir lokale Variablen, da sie nur in der Methode sichtbar sind. Die lokalen Variablen sind außerhalb der Methode nicht verfügbar. Innerhalb des Methodenrumpfes stehen Variablendeklarationen und Anweisungen, wie wir sie in Grundlagen der Informatik 1 kennengelernt haben.

```
public class Invoice
{
    ...
    public string ToString()
    {
        double netPrice = amount * (1 - discount);
        double grossPrice = netPrice * (1 + vat);
        return "An: " + recipient.ToString() +
            "\nNetto: " + netPrice +
            "\nMwSt-Satz: " + vat +
            "\nBrutto: "+grossPrice;
    }
}
```

Beachten Sie, dass diese Methode erst dann korrekt funktioniert, wenn wir eine Möglichkeit haben, den Rechnungsempfänger zu setzen. Zum jetzigen Zeitpunkt würde der Versuch, die Methode auszuführen, zu einer Fehlermeldung führen.

Bemerkung 2.2.7

In C# werden alle Variablen, egal ob es sich um lokale Variablen oder Attribute handelt, mit ihrem Standardwert initialisiert und können daher ohne explizite Initialisierung verwendet werden. In Java hingegen müssen lokale Variablen, nicht jedoch Attribute, explizit initialisiert werden, anderenfalls kompiliert das Programm nicht.

In der Methode setDiscount muss der im Parameter newDiscount übergebene Wert im Attribut discount gespeichert werden, so dass dieser nach der Methodenausführung im Zustand des Rechnungsobjekts gespeichert ist. Wir können in Methoden nicht nur den Wert von Attributen auslesen, sondern diesen auch verändern. Da Parameter lokale Variablen sind, erfolgt der Zugriff auf die gleiche Weise:

```
void setDiscount(double newDiscount)
{
    discount = newDiscount;
}
```

Einem Parameter kann wie jeder anderen lokalen Variable auch ein neuer Wert zugewiesen werden, jedoch sollte dies der Übersicht halber wenn möglich vermieden werden. Innerhalb von Methoden kann es nützlich sein, einen Verweis auf ein Objekt zu haben, an dem die Methode gerade aufgerufen wurde. Diese Referenz erhält man mit Hilfe des Schlüsselworts this. Der Typ der this-Referenz entspricht immer der umgebenden Klasse.

Bemerkung 2.2.8

Wenn in einer Methode eine lokale Variable den gleichen Namen wie ein Attribut trägt, so akzeptiert der Übersetzer dies, allerdings wird das Attribut innerhalb dieser Methode von der lokalen Variable verdeckt (engl. shadow).

Steht bei einer Namensgleichheit von Attribut und lokaler Variable kein this vor dem Zugriff, so erfolgt der Zugriff auf die lokale Variable. Mit Hilfe von this kann immer auf das Attribut zugegriffen werden. Bei Gettern und Settern, also Methoden, die nur dazu dienen, ein Attribut mit einem Wert zu belegen oder auszulesen, ist es üblich, dass man den Parameter genauso benennt wie das zu belegende Attribut. Der obige Setter setDiscount sieht mit dieser Konvention wie folgt aus:

```
void setDiscount(double discount)
{
    this.discount = discount;
}
```

Selbsttestaufgabe 2.2.8

Implementieren Sie die Rümpfe der beiden Methoden addPosition und setVat der Klasse Invoice aus Selbsttestaufgabe 2.2.4 .

Musterlösung zu Selbsttestaufgabe 2.2.8

```
public void addPosition(int quantity, double unitcost)
{
    this.amount += quantity * unitcost;
}
public void setVat(double vat)
{
    this.vat = vat;
}
```

Selbsttestaufgabe 2.2.9

Benennen Sie alle Attribute und lokale Variablen der Klasse A. Bei welchen lokalen Variablen handelt es sich um Parameter?

Bei welchen Zugriffen in der Methode x handelt es sich um Zugriffe auf lokale Variablen und bei welchen um Zugriffe auf Attribute?

```
1  class A
2  {
3      public double a;
4      public int b;
5      public string c;
6      public int d;
7
8      public void k(int y, double d)
9      {
10         int x = 10, z = 2;
11         double m = x * z / 3.0;
12     }
13     public void x(int a, int b)
14     {
15         double c = 3;
16         Console.WriteLine(this.c);
17         double f = a * this.b + c * b;
18         double g = this.a + d;
19     }
20 }
```

Musterlösung zu Selbsttestaufgabe 2.2.9

Attribute: a, b, c, d aus Zeilen 3-6
Lokale Variablen: y, d, x, m, z aus der Methode k und a, b, c, f, g aus der Methode x
Davon Parameter: y, d aus der Methode k und a, b aus der Methode x
Zugriffe in x *auf Attribute:* this.c in Zeile 16, this.b in Zeile 17, this.a in Zeile 18, d in Zeile 18. Die übrigen Zugriffe sind Zugriffe auf lokale Variablen.

Wenn eine Methode einen Ergebnistyp besitzt, so muss auch ein Wert an den Aufrufer zurückgegeben werden. Dies geschieht mit Hilfe einer return-Anweisung.

Definition 2.2.9 : return-Anweisung

Eine return-Anweisung im Rumpf einer Methode dient zwei Zwecken:

- Sie definiert einen Wert, dessen Typ verträglich sein muss mit dem Ergebnistyp der Methode in der sie vorkommt; dieser Wert ersetzt nach Ausführung der Methode den Methodenaufruf. Wenn die Methode den Rückgabetypen void hat, entfällt dieser Zweck.
- Sie gibt die Kontrolle an die Aufrufstelle zurück.

Nach einer return-Anweisung (im Sinne eines Ablaufpfads der Methode) werden keine weiteren Anweisungen mehr ausgeführt. In C# ist es zwar grundsätzlich nicht verboten, weiteren Code nach der return-Anweisung anzugeben, der Compiler warnt aber, dass er unerreichbaren Code entdeckt hat. Bei Java hingegen werden solche Anweisungen vom Compiler gar nicht erst akzeptiert.

Eine return-Anweisung hat die allgemeine Form:

```
return Ausdruck;
```

Wenn die return-Anweisung im Rumpf einer void-Methode benutzt wird, also einer Methode ohne Rückgabewert, darf kein Ausdruck in der return-Anweisung erscheinen. Andernfalls wird ein Übersetzerfehler auftreten. Stattdessen hat in diesem Fall eine return-Anweisung die Form:

```
return;
```

In einer void-Methode muss es keine return-Anweisung geben; in allen anderen Methoden muss jeder Berechnungszweig, das heißt jeder Weg durch den Code einer Methode, hingegen mit einer (geeigneten) return-Anweisung enden. Der Compiler führt keine komplizierten Tests durch, um zu ermitteln, ob eine Alternative logisch überhaupt möglich ist, das heißt, das Programmierer muss sicherstellen, dass jede Verzweigung, die durch eine alternative Auswahl (mit if oder switch) oder eine Schleife (mit while oder for) so fortgesetzt werden, dass man zu einer return-Anweisung gelangt – ungeachtet dessen ob die konkrete Abzweigungsfolge tatsächlich logisch möglich ist. Dass kein Versuch unternommen wird, die Verzweigungslogik genauer zu analysieren, liegt daran, dass es sich hierbei um ein unentscheidbares Problem handelt, vgl. Lektion 6.

Wollen wir nun eine Methode getDiscount implementieren so muss diese den aktuellen Wert des Attributs discount zurück liefern:

```
class Invoice
{
    double discount;
    ...
    public double getDiscount()
    {
        return discount;
    }
    ...
}
```

Wir haben bislang schon an einigen Stellen mit Gettern und Settern gearbeitet, ohne das Konzept ausdrücklich zu besprechen.

Bemerkung 2.2.10 : Getter und Setter

In den meisten Klassen haben wir Methoden, die das Auslesen und Verändern von Attributen ermöglichen, wie zum Beispiel die Methoden setDiscount und getDiscount. Üblicherweise beginnt der Name einer solchen Methode mit „get“ respektive „set“, daher nennt man die Methoden, die zum Auslesen von Attributen gedacht sind Getter und Methoden, die zum Verändern von Attributen gedacht sind, Setter.

Getter- und Setter-Methoden dienen der Wahrung des Geheimnisprinzips. Für die sichere Handhabung von Objekten einer nicht genauer bekannten Klasse ist es von Vorteil, wenn der Aufrufer nicht direkt auf die Attribute zugreift, sondern stattdessen Getter- und Setter-Methoden aufruft. Er muss dann nicht wissen, wie die Attribute intern implementiert sind. Das Verbergen von Klasseninterna ist das Hauptziel des Geheimnisprinzips. Selbst wenn in der Klassendefinition Veränderungen an der Implementierung der Attribute vorgenommen werden, wird der Aufrufer der Methode davon nicht beeinflusst. Setter ermöglichen es außerdem, sicherzustellen, dass den Attributen eines Objekts keine Werte zugewiesen werden, die einen inkonsistenten Zustand des Objekts zur Folge hätten.

In einem späteren Kapitel werden wir Mechanismen, die sogenannten Zugriffsmodifikatoren, kennen lernen, die den direkten Zugriff auf Attribute ganz unterbinden. Setter-Methoden können weiterhin so gestaltet werden, dass sie eine Überprüfung beinhalten, ob es sich bei dem neuen Wert für ein Attribut um einen zulässigen Wert handelt.

In C# ist der Zugriff auf Methoden und Attribute standardmäßig – d.h. ohne Zugriffsmodifikator – sehr streng geregelt, so dass wir bislang bereits auf einen Zugriffsmodifikator zurückgreifen mussten, nämlich public.

Selbsttestaufgabe 2.2.10

Ändern Sie die Methode setDiscount, so dass der gespeicherte Rabatt nur verändert wird, wenn der neue Rabatt nicht negativ und nicht größer als 50% ist.

Musterlösung zu Selbsttestaufgabe 2.2.10

```
void setDiscount(double discount)
{
    if(discount >= 0 && discount <=.5)
        this.discount = discount;
}
```

Selbsttestaufgabe 2.2.11

Ergänzen Sie die Klasse Invoice um Getter-Methoden für alle Attribute außer den

Rechnungsempfänger (recipient), noch fehlende Setter-Methoden, sowie um die nachfolgend beschriebenen Methoden.

- computeNetPrice: Errechnet den aktuellen Nettopreis, also Betrag abzüglich Rabatt und gibt diesen zurück.
- computeVat: Errechnet die Mehrwertsteuer, die für den aktuellen Nettopreis anfallen würde und gibt diese zurück.
- computeGrossPrice: Errechnet den aktuellen Bruttopreis, also Betrag abzüglich Rabatt zuzüglich Mehrwertsteuer und gibt diesen zurück.

Ergänzen Sie außerdem die Klasse Customer um Getter- und Setter-Methoden für die beiden Attribute name und address.

Musterlösung zu Selbsttestaufgabe 2.2.11

Wir geben nur die in der Aufgabe geforderten Methoden an:

```
public class Invoice
{
    #region attributes
    public Customer recipient;
    public double amount;
    public double discount;
    public double vat;
    #endregion
    #region gettersetter
    public void setAmount(double amount)
    {
        this.amount = amount;
    }
    public double getAmount()
    {
        return amount;
    }
    public void setDiscount(double discount)
    {
        if (discount >= 0 && discount <= .5)
            this.discount = discount;
    }
    public double getDiscount()
    {
        return discount;
```

```
    }
    public void setVat(double vat)
    {
        this.vat = vat;
    }
    public double getVat()
    {
        return discount;
    }
    #endregion
    #region methods
    public double computeNetPrice()
    {
        return amount * (1 - discount);
    }
    public double computeVat()
    {
        return computeNetPrice()*vat;
    }
    public double computeGrossPrice()
    {
        return computeNetPrice()+computeVat();
    }
    #endregion
}

public class Customer
{
    string name;
    string address;
    public string getName()
    {
        return name;
    }
    public string getAddress()
    {
        return address;
    }
    public void setName(string name)
    {
        this.name = name;
    }
    public void setAddress(string address)
```

```
    {
        this.address = address;
    }
}
```

Bisher haben wir in Methoden lediglich Attributzugriffe, return-Anweisungen und Anweisungen, die wir aus dem Kurs Grundlagen der Informatik 1 kennen, verwendet. Allerdings treten in einer Klasse oft an mehreren Stellen die gleichen Berechnungen auf. So haben wir sowohl in der Methode computeGrossPrice als auch in der Methode ToString den Bruttopreis berechnet. Um solche unnötigen Verdoppelungen von Anweisungen zu vermeiden, können wir in einer Methode auch andere Methoden aufrufen und deren Ergebnis zu einer weiteren Verarbeitung verwenden. Wir können somit die Methode ToString folgendermaßen anpassen:

```
public string ToString()
{
    return "An: " + recipient.ToString() +
        "\nNetto: " + computeNetPrice() +
        "\nMwSt-Satz: " + computeVat() +
        "\nBrutto: " + computeGrossPrice();
}
```

Steht vor einem Methodenaufruf nicht explizit ein Objekt, an dem sie aufgerufen werden soll, so erfolgt der Aufruf genau an dem Objekt, an dem schon die umgebende Methode aufgerufen wurde. Wir können auch ein this vor die Methodenaufrufe setzen. So wird deutlich, dass der Brutto- bzw. Nettopreis genau dieser Rechnung berechnet wird. Somit können auch direkte Zugriffe auf Attribute außerhalb der Getter- und Setter-Methoden vermieden werden. Statt

```
computeGrossPrice()
```

könnte man im obigen Code also beispielsweise auch schreiben

```
this.computeGrossPrice()
```

Selbsttestaufgabe 2.2.12

Gehen Sie den bisher besprochenen Code der Klasse Invoice durch und ersetzen Sie die direkten Zugriffe auf die Attribute durch Aufrufe der zugehörigen Getter.

Musterlösung zu Selbsttestaufgabe 2.2.12

Die folgenden Methoden müssen angepasst werden:

```
public void addPosition(int quantity, double unitcost)
{
```

```
    setAmount(getAmount() + quantity * unitcost);
}
public double computeNetPrice()
{
    return getAmount() * (1 - getDiscount());
}
public double computeVat()
{
    return computeNetPrice()*getVat();
}
```

Bisher waren die Typen der Parameter und Ergebnisse primitive Typen. Jedoch ist es auch möglich Referenzen auf Objekte als Parameter zu übergeben oder zurück zu liefern. Dies wird zum Beispiel nötig wenn wir Getter- und Setter-Methoden für das Attribut recipient implementieren.

```
public Customer getRecipient()
{
    return recipient;
}
public void setRecipient(Customer recipient)
{
    this.recipient = recipient;
}
```

Dieselben Gesetzmäßigkeiten, die für Zuweisungen gelten, gelten auch bei der Übergabe von Parametern an Methoden. Wird als Parameter ein Wert eines primitiven Datentyps übergeben, so hat die weitere Verarbeitung dieses Wertes durch die Methode keinerlei Auswirkungen auf die aufrufende Stelle. In diesem Fall spricht man von einem Aufruf mit Wertübergabe (engl. call by value). Wird einer Methode als Parameter ein Objekt übergeben, so handelt es sich, exakt gesprochen, um eine Referenz auf das Objekt. Sofern weitere Referenzen auf dieses Objekt existieren, was in der Regel zumindest an der aufrufenden Stelle der Fall ist, sind Veränderungen des Objekts an allen Referenzen gleichermaßen bemerkbar. Dies wird als Aufruf mit Verweisübergabe (engl. call by reference) bezeichnet.

Bemerkung 2.2.11

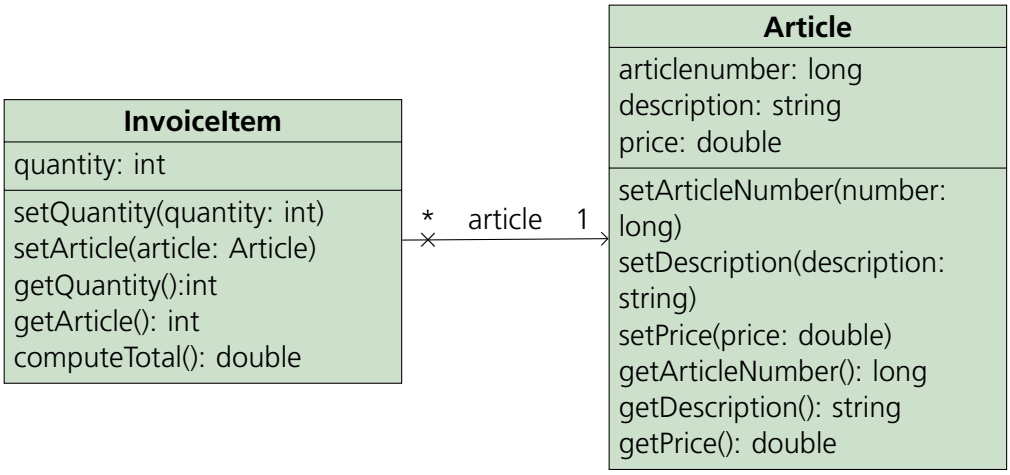
Wird dem Parameter eine andere Referenz zugewiesen, so hat dies keine Auswirkung auf den Aufrufer. Das gilt allerdings *nicht*, wenn der Parameter mit dem Schlüsselwort ref gekennzeichnet ist.

Wann immer eine Klasse in einer Methode Objekte verarbeitet (dies gilt ebenso für Parameter wie für eigene Attribute der Klasse), sollten Sie sich bewusst sein, dass weitere Referenzen auf diese Objekte existieren können, die von den Veränderungen mit betroffen sind.

Würde eine Methode der Klasse Invoice Manipulationen an dem durch ihr Attribut recipient referenzierten Customer-Objekt vornehmen, so sind diese Veränderungen auch außerhalb von Invoice-Objekten an anderen Referenzen auf dasselbe Customer-Objekt bemerkbar. Auch der umgekehrte Weg ist möglich. Es kann, falls außerhalb eines Invoice-Objekts eine entsprechende Referenz existiert, das von seinem Attribut recipient referenzierte Customer-Objekt manipuliert werden.

Selbsttestaufgabe 2.2.13

Implementieren Sie die im folgenden dargestellten Klassen Article und InvoiceItem.



Überladen Sie schließlich die Methode addPosition der Klasse Invoice so, dass sie als Parameter ein Objekt der Klasse InvoiceItem erwartet und an Hand der im InvoiceItem-Objekt gespeicherten Informationen den Betrag anpasst.

Musterlösung zu Selbsttestaufgabe 2.2.13

```
class InvoiceItem
{
    int quantity;
    Article article;
    public void setQuantity(int quantity)
    {
        this.quantity = quantity;
    }
```

```
    public int getQuantity()
    {
        return quantity;
    }
    public void setArticle(Article article)
    {
        this.article = article;
    }
    public Article getArticle()
    {
        return article;
    }
    public double computeTotal()
    {
        return article.getPrice() * getQuantity();
    }
}
```

```
class Article
{
    long articlenumber;
    string description;
    double price;
    public void setArticleNumber(long number)
    {
        articlenumber = number;
    }
    public long getArticleNumber()
    {
        return articlenumber;
    }
    public void setDescription(string description)
    {
        this.description = description;
    }
    public string getDescription()
    {
        return description;
    }
    public void setPrice(double price)
    {
        this.price = price;
    }
```

```
    public double getPrice()
    {
        return price;
    }
}
```

und in Invoice:

```
public void addPosition(InvoiceItem invoiceItem)
{
    setAmount(getAmount() + invoiceItem.computeTotal());
}
```

2.2.4 Konstruktordeklaration

Konstruktoren (engl. constructor) sind spezielle Methoden, um Instanzen einer Klasse zu erzeugen. Konstruktoren sind immer nach derjenigen Klasse benannt, deren Instanzen sie erzeugen. In ihnen werden in der Regel die nötigen Initialisierungen, zum Beispiel die Belegung der Attribute mit Anfangswerten, durchgeführt. Es handelt sich bei einem Konstruktor um eine besondere Methode, die immer das eben erzeugte Objekt zurück liefert. Ein Konstruktor wird nie an einem Objekt oder einer Klasse aufgerufen sondern immer in Zusammenhang mit dem Schlüsselwort new (sowohl in C# als auch in Java).

Definition 2.2.12 : Konstruktordeklaration

Eine Konstruktordeklaration besteht aus einem Kopf und einem Rumpf. Sie genügt folgender Form:

```
Konstruktorname(Parameterliste)
{
    Anweisungen
}
```

Der Name des Konstruktors muss mit dem Namen der Klasse, der er angehört, identisch sein. Anders als bei einer Methodendeklaration hat eine Konstruktordeklaration keinen expliziten Ergebnistyp und der Rumpf muss keine return-Anweisung enthalten. Es darf allerdings optional return-Anweisungen ohne Angabe eines Rückgabewertes enthalten, um die Ausführung der Methode sofort zu beenden.

Innerhalb eines Konstruktors kann auf die Elemente einer Klasse genau wie aus einer Methode, zum Beispiel mit this zugegriffen werden, auch ansonsten können im Rumpf die gleichen Anweisungen wie in einer Methode genutzt werden.

Eine Konstruktordeklaration mit einer leeren Parameterliste wird als Standardkonstruktor (engl. default constructor) bezeichnet.

Wird in einer Klasse kein Konstruktor explizit deklariert, so besitzt diese Klasse automatisch einen Standardkonstruktor mit leerem Rumpf. Das gilt sowohl in C# als auch in Java. In der Regel sollte jedoch eine Klasse eigene Konstruktoren deklarieren.

Wie Methoden haben auch Konstruktoren eine Signatur.

Definition 2.2.13 : Signatur eines Konstruktors

Die Signatur eines Konstruktors besteht aus dem Konstruktornamen sowie den Typen seiner formalen Parameter.

Konstruktorennamen können ebenso wie Methodennamen überladen werden. Eine Klasse kann also über mehrere Konstruktoren mit verschiedenen Parameterlisten verfügen.

Oftmals ist es sinnvoll, mehr als einen Konstruktor anzubieten. Um aber doppelten Quelltext in den einzelnen Konstruktoren zu vermeiden, kann ein Konstruktor auch genau einen anderen aufrufen. Gehen wir also davon aus, dass unsere Rechnung in der Regel einen Mehrwertsteuersatz von 19% besitzt. Wir bieten aber noch zusätzlich einen zweiten Konstruktor an, der den Mehrwertsteuersatz als Parameter erwartet. Ein Aufruf an einen anderen Konstruktor erfolgt mit Hilfe von this(Parameterliste). Ein solcher Aufruf muss, wenn vorhanden, immer die erste Anweisung in einem Konstruktor sein. In C# bedeutet das, dass dieser Aufruf noch vor dem Rumpf mit Doppelpunkt getrennt erfolgen muss, in Java ist es schlicht der erste Befehl im Rumpf.

```
public class Invoice
{
    #region attributes
    (...)
    public int number;
    #endregion
    #region constructors
    public Invoice(int number) : this(number, .19) { }
    public Invoice(int number, double vat)
    {
        this.vat = vat;
        this.number = number;
    }
    #endregion
                (...)
}
```

Eine weitere Möglichkeit, verschiedene Parameterlisten für Konstruktoren (oder Methoden im Allgemeinen) zuzulassen sind Standardwerte für Parameter. Diese gibt man an, indem man hinter den Parameter den Zuweisungsoperator setzt und dann den Standardwert angibt. In unserem Beispiel sähe das so aus:

```
public Invoice(int number, double vat=.19)
```

```
{
    this.vat = vat;
    this.number = number;
}
```

Der Vorteil dieser Lösung ist, dass der Tooltip in Visual Studio die Standardwerte mit angibt, es also ein wenig Transparenz darüber gibt, wie die Attribute initialisiert werden, denen man keinen expliziten Wert über den Konstruktor zuteilt. Die Lösung kann allerdings das Überladen nicht vollständig ersetzen, da insbesondere die Verwendung der Standardwerte nur von rechts nach links zulässig ist. Das heißt, wenn es drei Parameter gibt und man nur dem zweiten keinen Wert beim Aufruf des Konstruktors explizit zuweisen möchte, kann man hierzu nicht Standardwerte verwenden. In dem Beispiel kann man die Methode wahlweise mit zwei Parametern oder nur einem int-Parameter aufrufen.

Bemerkung 2.2.14

Die Instanzvariablen werden vor der Ausführung des Konstruktors initialisiert.

Selbsttestaufgabe 2.2.14

Ergänzen Sie die Klassen Customer, InvoiceItem und Article um geeignete Konstruktoren.

Musterlösung zu Selbsttestaufgabe 2.2.14

In Article:

```
public Article(long articlenumber=0, double price=0,
        string description="")
{
    setArticleNumber(articlenumber);
    setPrice(price);
    setDescription(description);
}
```

In InvoiceItem:

```
public InvoiceItem(Article article = null, int quantity = 1)
{
    this.quantity = quantity;
    this.article = article;
}
```

In Customer:

```
public Customer(string name="", string address="")
```

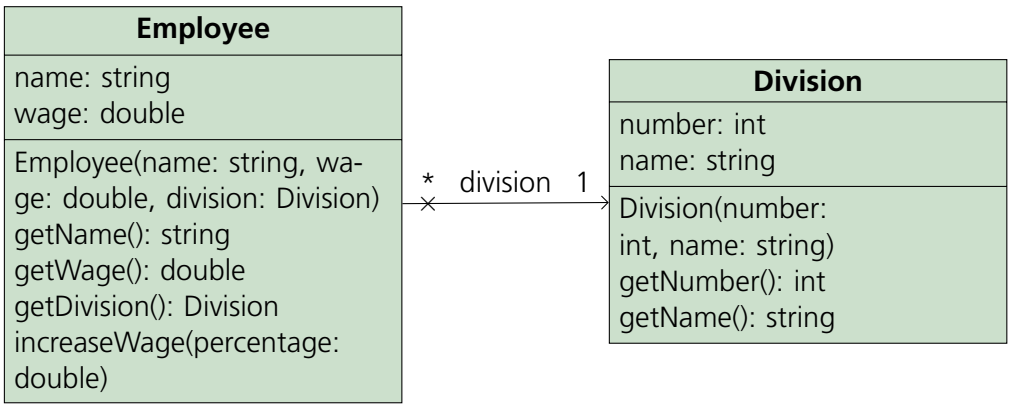
```
{
    this.name = name;
    this.address = address;
}
```

Selbsttestaufgabe 2.2.15

Lesen Sie die folgende Beschreibung gründlich und entwickeln Sie zunächst ein passendes UML-Klassendiagramm. Implementieren Sie die Klassen anschließend in C#:

Ein Angestelltes (Employee) besitzt einen Namen (name) und ein aktuelles Gehalt (wage). Zudem gehört jedes Angestelltes genau zu einer Abteilung (division). Eine Abteilung besitzt eine Nummer (number) und einen Namen (name). Name, Gehalt und Abteilung des Angestellten können erfragt und das Gehalt um einen gegebenen Prozentsatz erhöht werden. Alle Attribute können ausgelesen werden, aber es sind nach der Initialisierung durch den Konstruktor keine Änderungen außer die Erhöhung des Gehalts zulässig.

Musterlösung zu Selbsttestaufgabe 2.2.15




```
class Division
{
    int number;
    string name;

    public Division(int number, string name)
    {
        this.number = number;
        this.name = name;
    }
```

```
    public int getNumber()
    {
        return number;
    }
    public string getName()
    {
        return name;
    }
}
```

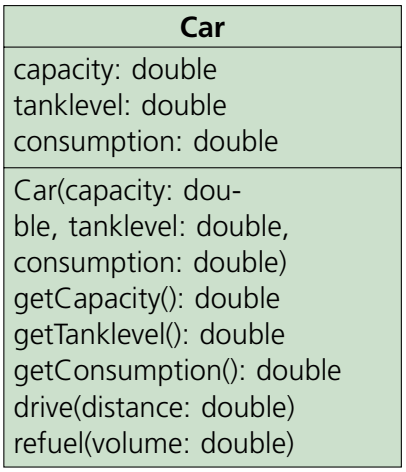
```
class Employee
{
    string name;
    double wage;
    Division division;

    public Employee(string name, double wage,
        Division division)
    {
        this.name = name;
        this.wage = wage;
        this.division = division;
    }
    public string getName()
    {
        return name;
    }
    public double getWage()
    {
        return wage;
    }
    public Division getDivision()
    {
        return division;
    }
    public void increaseWage(double percentage)
    {
        wage += wage * percentage;
    }
}
```

Selbsttestaufgabe 2.2.16

Entwickeln Sie zu der folgenden Beschreibung einer Klasse, die ein Auto repräsentiert, zunächst ein passendes UML-Klassendiagramm und implementieren Sie die Klassen anschließend in C#: Ein Auto hat einen Tank mit einer individuellen, maximalen Füllmenge. Ein Auto kann betankt werden. Zudem besitzt es einen durchschnittlichen Verbrauch (in Litern pro 100 Kilometer). Wenn das Auto eine Strecke gegebener Länge zurücklegt wird entsprechend Benzin verbraucht.

Implementieren Sie anschließend eine Main-Methode, in der ein neues Auto erzeugt wird und die verschiedenen Methoden ausprobiert werden.

Musterlösung zu Selbsttestaufgabe 2.2.16




```
class Car
{
    double capacity;
    double tanklevel;
    double consumption;

    public Car(double capacity, double tanklevel,
               double consumption)
    {
        this.capacity = capacity;
        this.tanklevel = tanklevel;
        this.consumption = consumption;
    }
    public double getCapacity()
    {
```

```
        return capacity;
    }

    public double getTanklevel()
    {
        return tanklevel;
    }
    public double getConsumption()
    {
        return consumption;
    }
    public void drive(double distance)
    {
        tanklevel -= distance * getConsumption() / 100;
        if (tanklevel < 0)
        {
            tanklevel = 0;
            Console.WriteLine("Nicht genug Treibstoff für
                die Distanz. Wagen ist auf der Strecke
                liegengeblieben.");
        }
    }
    public void refuel(double volume)
    {
        tanklevel += volume;
        if(tanklevel > getCapacity())
        {
            Console.WriteLine("Zu viel getankt. " +
                (tanklevel - getCapacity()) + " Liter
                wurden verschüttet.");
            tanklevel = getCapacity();
        }
    }
}
```

und in der Main-Methode:

```
Car car = new Car(275, 275, 5);
car.drive(20);
Console.WriteLine(car.getTanklevel()+"/"+car.getCapacity());
car.refuel(1000);
car.drive(500);
car.refuel(100);
Console.WriteLine(car.getTanklevel()+"/"+car.getCapacity());
```

2.3 Klassenvariablen und -methoden

Bisher haben wir nur Elemente der Klassendeklaration betrachtet, die es für jedes Objekt einer Klasse gibt oder die Objekte erzeugen. Jedoch können auch Klassen selbst bestimmte Eigenschaften besitzen oder von konkreten Objekten unabhängiges Verhalten anbieten. Klassenvariablen und Klassenmethoden sind Attribute und Methoden, die einer Klasse und nicht den Instanzen einer Klasse angehören. Im Gegensatz zu Instanzvariablen gibt es jede Klassenvariable genau einmal, unabhängig von der Anzahl der existierenden Instanzen der Klasse.

2.3.1 Klassenvariablen

Bisher haben wir die Rechnungsnummer im Konstruktor übergeben. Es wäre einfacher wenn diese Nummer automatisch erzeugt und keine Nummer doppelt vergeben werden würde. Die dafür notwendigen Informationen können in der dazugehörigen Klasse gespeichert werden. So können die Rechnungsnummern automatisch aufsteigend vergeben werden. Klassenvariablen können von den Objekten einer Klasse gemeinsam genutzt werden. Es handelt sich bei ihnen nicht um Eigenschaften eines konkreten Objekts, sondern um Eigenschaften der gesamten Klasse.

Definition 2.3.1 : Klassenvariable

Eine Klassenvariable wird durch das Schlüsselwort static, das der Attributvereinbarung vorangestellt wird, nach folgendem Muster deklariert:

```
static Typ Name
```

Bemerkung 2.3.2

Klassenvariablen werden auch als Klassenattribute bezeichnet. Um keine Verwirrung entstehen zu lassen, können Instanzvariablen dann auch als Objektattribute bezeichnet werden.

Soll von außerhalb einer Klasse auf eine Klassenvariable zugegriffen werden, so muss der Klassenname mit einem Punkt getrennt vorangestellt werden. In Java ist es alternativ zulässig, aber kein guter Stil, eine Klassenvariable auf einem Objekt anzusprechen, in C# hingegen wird das vom Compiler zurückgewiesen. Innerhalb der Klasse ist das Voranstellen des Klassennamens optional, kann aber der Transparenz dienlich sein.

Wir ergänzen die Klassenvariable nextNumber in der Klasse Invoice und initialisieren sie mit 10000. Im Konstruktor kann automatisch auf diese zugegriffen und das Objektattribut number entsprechend initialisiert werden. Allerdings darf dabei nicht vergessen werden den in nextNumber gespeicherten Wert anzupassen, so dass eine Nummer nicht mehrfach vergeben wird.

```
public class Invoice
{
    #region attributes
    (...)
    static int nextNumber = 10000;
    #endregion
    #region constructors
    public Invoice() : this(nextNumber++, .19) { }
    (...)
    #endregion
                (...)
}
```

Häufig werden in Klassen auch Konstanten benötigt, die für alle Methoden zur Verfügung stehen sollen oder auch für andere Klassen. Solche Konstanten werden in C# mit dem Schlüsselwort const wie Klassenattribute deklariert. In Java wird stattdessen das Schlüsselwort final verwendet. So könnten wir den Standardmehrwertsteuersatz als Konstante der Klasse Invoice deklarieren. Auf diese Konstante kann dann von außerhalb der Klasse mit dem Ausdruck Invoice.STANDARD_VAT zugegriffen werden. Wenn man eine Konstante definiert, muss man dieser in C# gleich bei der Deklaration einen Wert zuweisen. Dieser Wert muss vom Compiler auch unmittelbar als Konstante ausgewertet werden können, kann also beispielsweise nicht aus einer Berechnung aus variablen Werten hervorgehen. Benötigt man mehr Flexibilität, kann man alternativ das Schlüsselwort readonly verwenden, das es auch erlaubt, den Wert einer so definierten Variable einmalig im Konstruktor statt bei der Deklaration zu setzen. Der Vorteil der const-Variante ist allerdings, dass bereits beim Kompilieren die Konstante durch den entsprechenden Wert ersetzt wird und dadurch die Ausführung etwas beschleunigt wird.

Die besprochene Modifikation sähe in der Klasse Invoice so aus (angegeben sind nur die neuen oder modifizierten Methoden):

```
public class Invoice
{
    #region constants
    const double STANDARD_VAT = .19;
    #endregion
    #region constructors
    public Invoice() : this(nextNumber++, STANDARD_VAT) { }
    public Invoice(int number) : this(number, STANDARD_VAT) { }
    public Invoice(double vat) : this(nextNumber++, vat) { }
    public Invoice(int number, double vat)
    {
        this.vat = vat;
        this.number = number;
    }
    #endregion
```

```
}
```

Es ist eine Konvention, dass Konstanten in Großbuchstaben geschrieben werden und wenn sie aus mehreren Wörtern bestehen, diese mit einem Unterstrich (_) getrennt werden.

2.3.2 Klassenmethoden

Bei Objektattributen haben wir gelernt, dass auf diese nach Möglichkeit nur über ihre Getter- und Setter-Methoden zugegriffen werden sollte. Das gleiche gilt auch für Klassenattribute. Bei Ihnen erfolgt der Zugriff dann über Klassenmethoden. Eine solche Klassenmethode gehört zur Klasse und nicht zu einem konkreten Objekt. Sie kann somit auch nur auf Eigenschaften der Klasse, also Klassenattribute, und auf andere Klassenmethoden, nicht jedoch auf Objektmethoden oder Objektattribute zugreifen. In einer Klassenmethode ist somit auch keine this-Referenz verfügbar.

Definition 2.3.3 : Klassenmethoden

Klassenmethoden werden in C# und Java wie (Objekt-)Methoden vereinbart, mit dem Unterschied, dass ihnen das Schlüsselwort static vorangestellt ist.

Klassenmethoden können von außerhalb der Klasse – genau wie Klassenattribute – in C# nur angesprochen werden, indem man sie über den Klassennamen referenziert. In Java ist es zusätzlich möglich, sie über ein Objekt anzusprechen. In beiden Fällen ist innerhalb der Klasse, zu der die Klassenmethode gehört, auch der unqualifizierte Aufruf, das heißt nur über die Angabe des Methodennamens möglich.

Statt im Konstruktor die nächste Rechnungsnummer zu berechnen, können wir dafür eine eigene Methode implementieren.

```
public class Invoice
{
    #region constructors
    public Invoice(): this(computeNextNumber(), STANDARD_VAT) {}
    public Invoice(int number): this(number, STANDARD_VAT) {}
    public Invoice(double vat): this(computeNextNumber(), vat) {}
    public Invoice(int number, double vat)
    {
        this.vat = vat;
        this.number = number;
    }
    #endregion
    #region methods

    public static int computeNextNumber()
    {
        return nextNumber++;
```

```
    }
    (...)
    #endregion
}
```

Selbsttestaufgabe 2.3.1

Warum entstehen bei der Übersetzung der nachfolgenden Klasse Fehler?

```
class A
{
    static int x;
    int y;
    int z;
    static int f(int a, int y)
    {
        return a + 2 * y;
    }
    static int g(int y)
    {
        return 2 * this.y + A.x + z;
    }
    int h(int z)
    {
        return A.x + y + z;
    }
}
```

Musterlösung zu Selbsttestaufgabe 2.3.1

Die Methode g ist eine Klassenmethode, versucht aber, auf die this-Referenz zuzugreifen. Der Zugriff auf z wäre zudem auch ohne this-Referenz unzulässig, da es sich um ein Objektattribut handelt.

Mit unserem Wissen über Objekt- und Klassenmethoden sowie über Objekt- und Klassenvariablen können wir die Anweisung:

```
Console.WriteLine()
```

die wir so häufig benutzten, in ihre Namensbestandteile zerlegen. Console bezeichnet eine Klasse, die im C#-Namensraum System bereitgestellt wird und WriteLine ist eine Klassenmethode der Klasse Console.

Selbsttestaufgabe 2.3.2

Objektmethoden operieren (implizit oder explizit mit Hilfe der this-Referenz) auf den Attributen des jeweiligen Objekts, sie können aber auch auf die Klassenvariablen zugreifen. Erklären Sie, ob und, wenn ja, wie eine Klassenmethode auf Objektattribute zugreifen kann.

Musterlösung zu Selbsttestaufgabe 2.3.2

Direkt kann eine Klassenmethode das nicht tun, aber wenn die Klassenmethode ein Objekt der eigenen Klasse kennt – beispielsweise weil dieses der Methode übergeben wurde, ein Objekt als Klassenattribut existiert[a] oder weil ein Objekt in einer lokalen Variable angelegt wird, kann die Klassenmethode über dieses Objekt auf die Instanzvariable des Objekts zugreifen.

[a] Das kommt beispielsweise bei dem Designpattern Singleton vor, mehr dazu im Modul 63613: Moderne Programmiertechniken und Methoden.

2.4 Zusammenfassung der wesentlichen Lernziele der Lektion

Wenn Sie diese Lektion durchgearbeitet haben, sollten Sie die folgenden wichtigen Fragestellungen beantworten können:

- Was sind Objektattribute, Objektmethoden, Klassenattribute und Klassenmethoden?
- Wozu dienen die Schlüsselwörter this, static, const in C#?
- Was bedeutet Überladung?
- Wie funktionieren Standardparameter?
- Wie ruft man Methoden auf und was sind formale Parameter?
- Wie erzeugt man ein Objekt?
- Wie kann man bei einem überladenen Konstruktor einen anderen Konstruktor aufrufen?
- Was sind Wert- und Referenzvariablen?

Sie sollten außerdem die folgenden Techniken beherrschen:

- Sie können eine Klasse mit allen wesentlichen Bestandteilen – Attribute, Konstruktoren, Methoden – deklarieren

- Sie können Objekte einer Klasse manipulieren.
- Sie können nachvollziehen ob ein Objekt zur Freigabe über den Garbage Collector freigegeben ist
- Sie können einfache Klassendiagramme bestehend aus Klassen, Attributen, Methoden und Assoziationen zwischen Klassen umsetzen.
- Sie können Getter und Setter definieren und erläutern, wozu diese dienen.

Lektion 4: Algorithmen

In den bisherigen Lektionen haben wir uns auf die Strukturierung des Quellcodes konzentriert, die betrachteten Methoden sind aber bislang von eher einfacher Natur. In dieser Lektion werden wir besprechen, wie man komplexere Operationen durchführt. Wir werden uns in dieser Lektion damit beschäftigen wie wir in Feldern suchen und diese sortieren können. Anschließend lernen wir das Prinzip der Rekursion kennen, das uns eine andere Herangehensweise für den Algorithmenentwurf bietet. Auch hierbei werden wir wieder auf Suchen und Sortieren von Feldern zurückkommen.

Ausgehend von der Beobachtung, dass bei rekursiven Lösungen Aufwand oft mehrfach anfällt, werden wir sehen, wie man mit dynamischer Programmierung den Rechenaufwand reduzieren kann. Dabei machen wir von der grundlegenden Methode der Optimierung des Rechenaufwands auf Kosten des Speicherbedarfs Gebrauch. Zum Abschluss geben wir einen kurzen Überblick über die Grundlagen der Kryptografie, indem wir die Klassifikation in symmetrische und asymmetrische Verschlüsselungsverfahren allgemein und jeweils an einem Beispiel besprechen.

4.1 Suchen und Sortieren

Bisher haben wir uns primär mit einfachen Klassen und ihrem Entwurf beschäftigt und Techniken zur Qualitätssicherung kennen gelernt. Für reale Anwendungen werden jedoch meistens komplexere Daten und komplexeres Verhalten benötigt. Deshalb werden wir uns in diesem Kapitel mit gängigen Problemen und Algorithmen zum Thema Suchen und Sortieren beschäftigen. Diese Verfahren können auf verschiedenen Datenstrukturen angewendet werden. Wir werden uns in diesem Kapitel zunächst auf Felder beschränken; weitere Datenstrukturen werden wir in der nächsten Lektion vorstellen.

4.1.1 Suchen in Feldern

Suchen in strukturierten Datensammlungen ist eine häufig auftretende Programmieraufgabe, zu deren Lösung zahlreiche Algorithmen entwickelt wurden. Zunächst wollen wir eine Methode implementieren, die prüft, ob ein bestimmter Wert ist einem Feld enthalten ist. Dazu vergleichen wir jedes Element mit dem gesuchten Wert.

```
public bool contains(int value, int[] array)
{
    foreach(int i in array)
```

```
    {
        // Ist aktueller Kandidat i gesuchter Wert value?
        if (value == i)
            // wenn Wert gefunden, Methode beenden
            return true;
    }
    // Alle Kandidaten wurden überprüft, Wert nicht gefunden
    return false;
}
```

Wir können die Methode sofort verlassen, wenn wir den gesuchten Wert gefunden haben. Wird das Ende der Schleife erreicht, so wurde der Wert nicht gefunden.

Selbsttestaufgabe 4.1.1

Implementieren Sie eine Methode `int countOccurances(int value, int[] array)`, die zählt, wie oft der übergebene Wert `value` in dem Feld `array` enthalten ist.

Musterlösung zu Selbsttestaufgabe 4.1.1

```
public int countOccurances(int value, int[] array)
{
    int n = 0;
    foreach (int i in array)
    {
        // Ist aktueller Kandidat i gesuchter Wert?
        if (value == i)
            // wenn Wert gefunden, Zähler erhöhen
            ++n;
    }
    // Alle Kandidaten wurden überprüft, Zähler zurückgeben
    return n;
}
```

Bei unserer Implementierung von `contains` haben wir keine Annahmen über eine mögliche Sortierung der Einträge im Feld getroffen. Wenn wir allerdings wissen, dass die Einträge im Feld sortiert sind, ist es nicht notwendig, das gesamte Feld zu durchlaufen. Stattdessen können wir unter der Annahme einer aufsteigenden Sortierung der Feldelemente den Durchlauf durch das Feld abbrechen, sobald wir einen Eintrag finden, der größer ist als der gesuchte Eintrag. In diesem Fall ergibt sich die folgende Implementation:

```
public bool containsSorted(int value, int[] array)
```

```
{
    foreach(int i in array)
    {
        // Ist aktueller Kandidat i gesuchter Wert value?
        if (value == i)
            // Wenn Wert gefunden, Methode beenden.
            return true;
        // Ist aktueller Kandidat zu groß?
        if (value < i)
            // Suche kann abgebrochen werden.
            return false;
    }
    // Alle Kandidaten wurden überprüft, Wert nicht gefunden
    return false;
}
```

Felder können nicht nur auf identische Elemente durchsucht werden, sondern auch auf Objekte, die bestimmte Eigenschaften aufweisen. So könnten beispielsweise alle Rechnungen eines bestimmten Kunden gesucht werden oder der Gesamtbetrag aller Rechnungen eines Kunden bestimmt werden.

Selbsttestaufgabe 4.1.2

Gegeben seien die folgenden Implementierungen; angegeben sind nur die relevanten Codeteile für diese Aufgabe:

```
public class Customer
{
}
```

```
public class Invoice
{
    private Customer recipient;
    public Customer getRecipient()
    {
        return recipient;
    }
    public int getSum()
    {
        (...)
    }
}
```

```
class InvoiceCollection
{
    private Invoice[] invoices;
    public int getSumOfAllInvoicesFor(Customer c)
    {
        (...)
    }
}
```

Implementieren Sie die Methode getSumOfAllInvoicesFor, die die Summe aller Rechnungsbeträge des übergebenen Kunden c (in Cent) berechnet.

Musterlösung zu Selbsttestaufgabe 4.1.2

```
public int getSumOfAllInvoicesFor(Customer c)
{
    int sum = 0;
    foreach(Invoice i in invoices)
    {
        if (i.getRecipient() == c) sum += i.getSum();
    }
    return sum;
}
```

Mit sehr ähnlichen Mitteln kann man auch das Maximum oder Minimum in einem (nicht notwendigerweise sortierten) Feld ermitteln.

Selbsttestaufgabe 4.1.3

Implementieren Sie eine Methode Invoice findMostExpensiveInvoice() in der Klasse InvoiceCollection, die die Rechnung mit dem größten Betrag bestimmt.

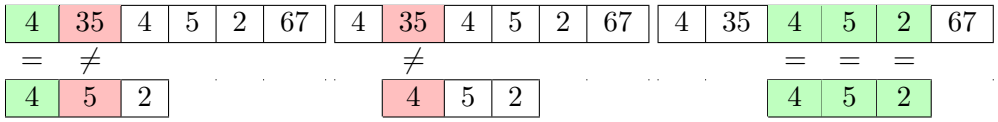
Musterlösung zu Selbsttestaufgabe 4.1.3

```
public Invoice findMostExpensiveInvoice()
{
    //Variablenrolle: Akkumulator
    int maxsum = 0;
    Invoice mostExpensive = null;
    foreach (Invoice i in invoices)
    {
```

```
            if (i.getSum() >= maxsum)
            {
                maxsum = i.getSum();
                mostExpensive = i;
            }
        }
        return mostExpensive;
}
```

Alle diese Suchen können auch auf mehrdimensionalen Feldern ausgeführt werden – wenn wie in den Beispielen die foreach-Schleife verwendet wird, sogar ohne die Notwendigkeit einer Anpassung der Suchschleife.

Bisher haben wir einzelne Elemente betrachtet, doch auch Teilfolgen können in einem Feld gefunden werden. So kann beispielsweise geprüft werden, ob eine bestimmte Zahlenfolge in einem Feld vorkommt. Man beginnt damit, den ersten Wert des Feldes mit dem ersten Wert der Zahlenfolge zu vergleichen. Stimmen die Werte überein, vergleicht man anschließend die zweiten Werte und so weiter. Stimmen die Werte nicht überein, so weiß man lediglich, dass die Zahlenfolge nicht beim ersten Element des Feldes beginnend gefunden werden kann. Man beginnt dann mit dem zweiten Element des Feldes und vergleicht es mit dem ersten der gesuchten Folge. Stimmen diese überein, so vergleicht man anschließend das dritte mit dem zweiten und so weiter.



Wurde die gesamte Teilfolge gefunden, so kann die Methode beendet werden. Bei der Implementierung müssen wir aufpassen, dass wir nicht auf ungültige Feldelemente zugreifen.

```
public bool containsSubsequence(int[] array, int[] subsequence)
{
    for (int i = 0; i <= array.Length - subsequence.Length; ++i)
        for (int j = 0; j < subsequence.Length; ++j)
            if (array[i+j] != subsequence[j])
                // Zahl an Position j stimmt nicht überein
                // Suche Teilfolge neu ab Index i+1
                break;
            else if(j == subsequence.Length-1)
                // Gesamte Teilfolge gefunden
                return true;
    // Teilfolge nicht gefunden
    return false;
}
```

Selbsttestaufgabe 4.1.4

Implementieren Sie eine Methode findBeginningOfWord(char[] array, String word), die das Wort word im Feld array sucht und den Index zurückgibt, an dem word startet. Falls das Wort im Feld nicht vorhanden ist, soll -1 zurückgegeben werden.

Musterlösung zu Selbsttestaufgabe 4.1.4

```
public int findBeginningOfWord(char[] array, string word)
{
    for (int i = 0; i < array.Length - word.Length + 1; ++i)
        for (int j = 0; j < word.Length; ++j)
            if (array[i + j] != word[j])
                //Zeichen an Position j stimmt nicht überein
                //Suche Wort neu ab Index i+1
                break;
            else if (j == word.Length - 1)
                //Gesamtes Wort gefunden
                return i;
    //Teilfolge nicht gefunden
    return -1;
}
```

Bemerkung 4.1.1

Das in Selbsttestaufgabe 4.1.4 beschriebene Problem ist als String-Matching-Problem bekannt. Zur Lösung dieses Problem gibt es eine Vielzahl von Algorithmen. Ein effizienteres Verfahren für ein festes Zielwort nutzt deterministische endliche Automaten, die wir in Lektion 7 kennenlernen werden.

Nachdem wir uns bisher mit der Suche in Feldern beschäftigt haben, werden wir uns im nächsten Abschnitt der Sortierung von Feldern widmen. Die beiden Probleme hängen insofern zusammen, als eine Suche auf sortierten Daten meistens effizienter lösbar ist.

4.1.2 Sortieren von Feldern

Das Suchen in großen Datenbeständen wird erheblich beschleunigt, wenn die Datenelemente gemäß einer Ordnungsrelation sortiert wurden. Der Suchalgorithmus kann diese Eigenschaft nutzen, um ein bestimmtes Element in der sortierten Menge zu finden. Beim Kartenspielen sortieren wir in der Regel unser Blatt nach dem Verfahren „Sortieren beim

Einfügen“ (engl. insertion sort). Wir nehmen eine Karte auf und fügen sie an der richtigen Stelle des bereits sortierten Blatts ein. Dieses Verfahren können wir auch beim Sortieren der Elemente eines Feldes nachbilden. Wir betrachten das erste Element des Feldes als sortierte und den Rest der Elemente als unsortierte Menge. Nun vergleichen wir, mit dem zweiten Element beginnend, die nicht sortierten Elemente mit den Elementen mit kleinerem Index (die ja bereits sortiert sind) und fügen das betrachtete Element durch Vertauschen (engl. swap) von rechts nach links an der richtigen Stelle ein. Das Feld ist sortiert, wenn das Ende des Feldes erreicht ist.

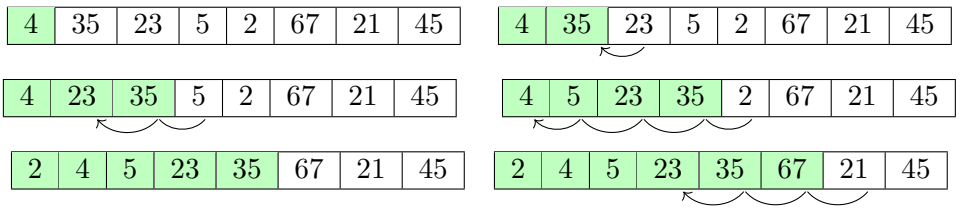
Algorithmus 4.1.2 : Insertion Sort

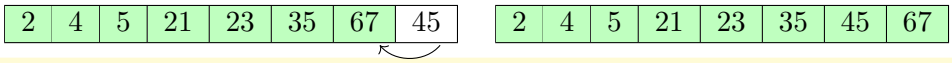
```
public void insertionSort(int[] array)
{
    //Beginne beim zweiten Element und betrachte
    //das Array bis zum Index i-1 als sortiert
    for(int i=1; i<array.Length; ++i)
        //gehe solange von i nach links bis das
        //Element an der richtigen Stelle steht.
        for(int j=i; j>0; --j)
            if (array[j - 1] > array[j])
            {
                //Wenn linkes größer, vertausche Elemente
                int temp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = temp;
            }
            else
                //Das Element ist an der korrekten Stelle.
                break;
}
```

Die C#-Implementation von Insertion Sort manipuliert das übergebene Feld direkt und hat daher keinen Rückgabewert.

Beispiel 4.1.3

Wir führen Insertion Sort beispielhaft an dem Feld $[4, 35, 23, 5, 2, 67, 21, 45]$ aus:





Selbsttestaufgabe 4.1.5

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus Insertion Sort, um das folgende Feld zu sortieren?

$$[4, 35, 23, 5, 2, 67, 45, 21]$$

Als Vergleich zählt die Auswertung der if-Bedingung. Wir betrachten die drei Anweisungen im if-Block als eine Vertauschungsoperation.

Musterlösung zu Selbsttestaufgabe 4.1.5

Es werden 18 Vergleiche und 12 Vertauschungen benötigt.

Selbsttestaufgabe 4.1.6

Modifizieren Sie die Insertion Sort-Implementation so, dass sie das übergebene Array absteigend statt aufsteigend sortiert.

Musterlösung

```
public void insertionSortDescending(int[] array)
{
    //Beginne beim zweiten Element und betrachte
    //das Array bis zum Index i-1 als sortiert
    for(int i=1; i<array.Length; ++i)
        //gehe solange von i nach links bis das
        //Element an der richtigen Stelle steht.
        for(int j=i; j>0; --j)
            if (array[j - 1] < array[j])
            {
                //Wenn linkes kleiner, vertausche die Elemente
                int temp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = temp;
            }
            else
```

```
                //Das Element ist an der korrekten Stelle.
                break;
}
```

Ein weiterer klassischer Sortieralgorithmus ist Bubblesort. Bubblesort gleicht dem Sortieren durch Einfügen darin, dass benachbarte Elemente vertauscht werden, sofern sie nicht geordnet sind. Bubblesort vollzieht diese Vertauschungen aber in einer anderen Reihenfolge. Der Algorithmus ist der Beobachtung nachgebildet, dass sich verschieden große, in einer Flüssigkeit aufsteigende Blasen von selbst sortieren, weil die kleineren Blasen von den größeren beim Aufsteigen überholt werden. Der Algorithmus vergleicht zwei aufeinander folgende Feldelemente feld[i] und feld$[i+1]$ miteinander. Falls feld$[i] >$ feld$[i+1]$ gilt, werden die Werte der beiden Elemente miteinander vertauscht. Beginnend mit dem ersten Feldelement wird das Feld durchlaufen. Wenn nötig werden die beiden Elemente vertauscht. Dies wird solange von vorne wiederholt, bis keine Vertauschungen mehr notwendig sind, weil kein größerer Wert mehr vor einem kleineren vorkommt.

Bubblesort stellt in jedem Durchlauf sicher, dass das größte bislang noch nicht sortierte Element an den korrekten Platz im Array kommt. Nach dem ersten Durchlauf steht also sicher das größte Element am Ende des Arrays, nach dem zweiten Durchlauf steht das zweitgrößte Element an vorletzter Stelle und so weiter.

Algorithmus 4.1.4 : Bubblesort

```
public void bubbleSort(int[] array)
{
    for (int i = 0; i < array.Length - 1; ++i)
    {
        //Sobald in einem Durchlauf keine Vertauschungen
        //vorgenommen werden ist das Feld sortiert.
        //(Variablenrolle: Flag)
        bool sorted = true;
        // Durchlaufe das Feld, in jedem Durchlauf muss
        // ein Element weniger berücksichtigt werden
        for (int j = 0; j < array.Length - 1 - i; j++)
        {
            if (array[j] > array[j + 1])
            {
                //Wenn linkes größer dann vertausche
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
```

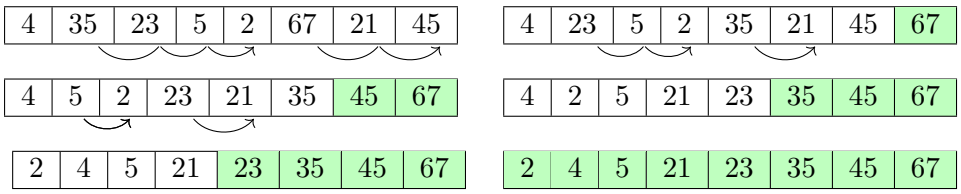
```
                    //Merken, dass eine Vertauschung
                    //durchgeführt wurde.
                    sorted = false;
                }
            }
            if(sorted) break;
        }
    }
```

Die C#-Implementation von Bubblesort manipuliert das übergebene Feld direkt und hat daher keinen Rückgabewert.

Wir illustrieren Bubblesort analog zu Insertion Sort an einem Beispiel:

Beispiel 4.1.5

Wir sortieren ein weiteres Mal das Feld $[4, 35, 23, 5, 2, 67, 21, 45]$, dieses Mal mit Bubblesort:



Wir schließen den Abschnitt mit einigen Selbsttestaufgaben:

Selbsttestaufgabe 4.1.7

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus Bubblesort, um das folgende Feld zu sortieren?

$$[4, 35, 23, 5, 2, 67, 45, 21]$$

Als Vergleich zählt die Auswertung der if-Bedingung.

Wir betrachten die drei Anweisungen im inneren if-Block als eine Vertauschungsoperation.

Musterlösung zu Selbsttestaufgabe 4.1.7

Es werden 25 Vergleiche und 12 Vertauschungen benötigt.

Selbsttestaufgabe 4.1.8

Entwerfen Sie, indem Sie den Bubblesort-Algorithmus modifizieren, eine Methode void sortAscending(Invoice[] invoices), die die Rechnungen aufsteigend nach ihren Beträgen sortiert.

Musterlösung zu Selbsttestaufgabe 4.1.8

```
public void sortAscending(Invoice[] invoices)
{
    for (int i = 0; i < invoices.Length - 1; ++i)
    {
        bool sorted = true;
        for (int j = 0; j < invoices.Length - 1 - i; j++)
        {
            if (invoices[j].getSum()>invoices[j+1].getSum())
            {
                Invoice temp = invoices[j];
                invoices[j] = invoices[j + 1];
                invoices[j + 1] = temp;
                sorted = false;
            }
        }
        if(sorted) break;
    }
}
```

Selbsttestaufgabe 4.1.9

Der Algorithmus „Sortieren durch Auswählen“ (engl. selection sort) nimmt am Anfang das komplette Feld als unsortiert an, sucht sich das größte Element unter den unsortierten Elementen und vertauscht es, wenn nötig, anschließend mit dem letzten Element des unsortierten Bereichs.

Im nächsten Schritt gehört das letzte Element nun zum sortierten Bereich. Es wird wiederum das größte Element des unsortierten Bereichs mit dem letzten Element vertauscht, so dass es anschließend zum sortierten Bereich gehört. Der Algorithmus terminiert, wenn der unsortierte Bereich aus einem Element besteht.

Wenden Sie den Algorithmus auf das folgende Feld an und zählen Sie die dabei

nötigen Vergleichs- und Vertauschungsoperationen.

$$[4, 35, 23, 5, 2, 67, 45, 21]$$

Musterlösung zu Selbsttestaufgabe 4.1.9

Es werden 28 Vergleiche und 6 Vertauschungen benötigt.

Selbsttestaufgabe 4.1.10

Implementieren Sie die Methode void selectionsort(int[] array) so, dass sie das übergebene Feld mit Hilfe des Algorithmus „Sortieren durch Auswählen" sortiert.

Musterlösung zu Selbsttestaufgabe 4.1.10

```
public void selectionSort(int[] array)
{
    for (int i = 0; i < array.Length; ++i)
    {
        int maximum = array[0];
        int maxIndex = 0;
        //Finde Maximum in unsortiertem Teilfeld
        for (int j = 1; j < array.Length - i; ++j)
        {
            if (array[j] > maximum)
            {
                maximum = array[j];
                maxIndex = j;
            }
        }
        //Tausche letztes Element mit Maximum falls
        //es nicht schon das Maximum ist.
        if (maxIndex != array.Length - i - 1)
        {
            array[maxIndex] = array[array.Length - i - 1];
            array[array.Length - i - 1] = maximum;
        }
    }
}
```

Selbsttestaufgabe 4.1.11

Vergleichen Sie die bei den verschiedenen Sortieralgorithmen (Insertion Sort, Bubblesort, Selection Sort) notwendigen Vergleichs- und Vertauschungsoperationen für die folgenden Felder:

$$a_1 = [4, 35, 23, 5, 2, 67, 45, 21]$$

$$a_2 = [2, 4, 5, 21, 23, 35, 45, 67]$$

$$a_3 = [67, 45, 35, 23, 21, 5, 4, 2]$$

Was fällt Ihnen dabei auf?

Musterlösung zu Selbsttestaufgabe 4.1.11

Bubblesort		
Feld	Vergleiche	Vertauschungen
a_1	25	12
a_2	7	0
a_3	28	28

Insertion Sort		
Feld	Vergleiche	Vertauschungen
a_1	18	12
a_2	7	0
a_3	28	28

Selection Sort		
Feld	Vergleiche	Vertauschungen
a_1	28	6
a_2	28	0
a_3	28	4

Es fällt auf, dass Sortieren durch Auswählen weniger Vertauschungsoperationen als die anderen Verfahren benötigt, jedoch immer gleich viele Vergleiche. Die anderen beiden Verfahren schneiden dafür bei schon sortierten Feldern besser ab, sind jedoch bei umgekehrt sortierten Feldern deutlich langsamer. Beispielhaft sei der Algorithmus Sortieren durch Auswählen einmal anhand von Beispiel 3 illustriert. Ein senkrechter Strich markiert jeweils den Beginn des sortierten Bereichs im Feld: Ausgangssituation:
$[67, 45, 35, 23, 21, 5, 4, 2]$
größtes Element: 67, Tausch: (1)
$[2, 45, 35, 23, 21, 5, 4|, 67]$
größtes Element: 45, Tausch: (2)
$[2, 4, 35, 23, 21, 5|, 45, 67]$
größtes Element: 35, Tausch: (3)
$[2, 4, 5, 23, 21|, 35, 45, 67]$
größtes Element: 23, Tausch: (4)
$[2, 4, 5, 21|, 23, 35, 45, 67]$
größtes Element: 21, kein Tausch:
$[2, 4, 5|, 21, 23, 35, 45, 67]$

größtes Element: 5, kein Tausch:
$[2, 4|, 5, 21, 23, 35, 45, 67]$
größtes Element: 4, kein Tausch:
$[2|, 4, 5, 21, 23, 35, 45, 67]$
fertig sortiert

4.2 Rekursion

Bisher haben wir Algorithmen auf der Basis von Programmstrukturen, die aus Folgen von Anweisungen, aus Verzweigungen und aus Schleifen bestanden, entwickelt. Schleifen wurden insbesondere dann verwendet, wenn Datenstrukturen mit einer variierenden Anzahl von Elementen iterativ zu verarbeiten waren. In diesem Kapitel wollen wir unser Inventar an Programmstrukturen um das Prinzip der Rekursion erweitern.

Definition 4.2.1 : Rekursiver Algorithmus

Ein Algorithmus ist rekursiv, wenn er Methoden (oder Funktionen) enthält, die sich selbst aufrufen. Eine Methode, die sich selbst aufruft, nennen wir eine rekursive Methode. Eine rekursive Methode umfasst zwei grundlegende Teile:

- den Basisfall, für den das Problem auf einfache Weise gelöst werden kann, und
- die rekursive Definition.

Die rekursive Definition besteht aus drei Facetten:

1. die Aufteilung des Problems in einfachere Teilprobleme,
2. die rekursive Anwendung auf alle Teilprobleme und
3. die Kombination der Teillösungen in eine Lösung für das Gesamtproblem.

Bei der rekursiven Anwendung ist darauf zu achten, dass wir uns mit zunehmender Rekursionstiefe an den Basisfall annähern. Wir bezeichnen diese Eigenschaft als Konvergenz der Rekursion.

Eine rekursive algorithmische Lösung bietet sich immer dann an, wenn man ein Problem in einfachere Teilprobleme aufspalten kann, die mit dem Originalproblem strukturell identisch, aber hinsichtlich einer passenden Kenngröße kleiner sind und die man zuerst nach dem gleichen Verfahren löst.

Beispiel 4.2.2

Nehmen wir an, dass wir die Summe der ersten n natürlichen Zahlen berechnen wollen. Dies können wir mit unserem bisherigen Wissen iterativ mit Hilfe einer Schleife

lösen:

```
public int sumIterative(int n)
{
    int result = 0;
    for (int i = 1; i <= n; ++i)
        result += i;
    return result;
}
```

Wir können es aber auch als rekursives Problem definieren, denn die Summe der ersten n Zahlen lässt sich berechnen, indem zunächst die Summe der ersten $n-1$ Zahlen berechnet wird und anschließend die n-te Zahl hinzu addiert wird. Das Problem wird dadurch von n Zahlen auf $n-1$ Zahlen reduziert. Natürlich müssen wir auch einen Basisfall identifizieren. Dieser ist erreicht, wenn keine Zahl mehr übrig ist. Wir können das Problem folgendermaßen mathematisch beschreiben:

$$\text{summe}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ \text{summe}(n-1) + n & \text{sonst} \end{cases}$$

In C# können wir diesen Rückgriff auf dieselbe Funktion umsetzen, indem wir dieselbe Methode erneut aufrufen:

```
public int sumRecursive(int n)
{
    if (n == 0) return 0;
    else return sumRecursive(n - 1) + n;
}
```

Beim Aufruf sumRecursive(5) finden die folgenden Berechnungen statt:
sumRecursive(5) =
sumRecursive(5 − 1) + 5 =
(sumRecursive(4 − 1) + 4) + 5 =
((sumRecursive(3 − 1) + 3) + 4) + 5 =
(((sumRecursive(2 − 1) + 2) + 3) + 4) + 5 =
((((sumRecursive(1 − 1) + 1) + 2) + 3) + 4) + 5 =
((((0 + 1) + 2) + 3) + 4) + 5 =
(((1 + 2) + 3) + 4) + 5 =
((3 + 3) + 4) + 5 =
(6 + 4) + 5 =
10 + 5 =
15

Selbsttestaufgabe 4.2.1

Was passiert, wenn die Methode sumRecursive mit negativen Werten aufgerufen wird? Wie kann dieses Problem gelöst werden?

Musterlösung zu Selbsttestaufgabe 4.2.1

Es wird eine StackOverflowException ausgelöst. Das kann man vermeiden, indem man zusätzlich überprüft, ob die übergeben Zahl negativ ist:

```
public int sumRecursiveNeg(int n)
{
    if (n == 0) return 0;
    else if(n < 0) return sumRecursiveNeg(n + 1) + n;
    return sumRecursiveNeg(n - 1) + n;
}
```

Ein weiteres Beispiel ist die Fakultätsfunktion. Sie spielt bei vielen mathematischen Anwendungen eine wichtige Rolle. Sie wird typischerweise durch das Ausrufezeichen „!“ symbolisiert. Die Fakultätsberechnung ist für Eingabewerte $n \geq 0$ wie folgt (links rekursiv, rechts iterativ) definiert:

$$n! = \begin{cases} n \cdot (n-1)! & \text{falls } n > 1 \\ 1 & \text{sonst} \end{cases} \qquad n! = \prod_{i=1}^{n} i$$

Selbsttestaufgabe 4.2.2

Implementieren Sie eine Methode facultyIterative(int n), die iterativ die Fakultät berechnet und eine Methode facultyRecursive(int n), die die Fakultät rekursiv berechnet. Die Methoden sollen dabei nur Werte $n \geq 0$ akzeptieren und anderenfalls eine geeignete Ausnahme werfen.

Musterlösung zu Selbsttestaufgabe 4.2.2

```
public int facultyRecursive(int n)
{
    if (n < 0) throw new ArgumentException();
    if (n == 0) return 1;
    else return facultyRecursive(n - 1) * n;
}
```

```
public int facultyIterative(int n)
{
    if (n < 0) throw new ArgumentException();
    int result = 1;
    for (int i = 1; i <= n; ++i) result *= i;
    return result;
}
```

Selbsttestaufgabe 4.2.3

Begründen Sie, wieso Ihre rekursive Implementation aus Selbsttestaufgabe 4.2.2 konvergiert.

Musterlösung zu Selbsttestaufgabe 4.2.3

Für alle $n < 0$ und für $n = 0$ terminiert die Methode offensichtlich direkt. Für alle übrigen n, also $n > 0$ wird die Lösung auf $n - 1$ zurückgeführt. Nach n rekursiven Aufrufen wird also das Rekursionsende erreicht und daher konvergiert die Methode.

Selbsttestaufgabe 4.2.4

Schreiben Sie die Methode int power(int p, int n), die für zwei ganze Zahlen p und n rekursiv den Wert p^n berechnet. Für den Fall $p = n = 0$ soll eine geeignete Exception geworfen werden.

Musterlösung zu Selbsttestaufgabe 4.2.4

```
public double power(int p, int n)
{
    if (n == 0)
        if (p == 0) throw new ArgumentException();
        else return 1;
    if(n > 0) return p * power(p, n-1);
    return power(p, n + 1) / p;
}
```

Die bisher betrachteten rekursiven Methoden hatten die Eigenschaft, dass sie nur einen rekursiven Aufruf pro Ablaufpfad beinhalten. Allerdings gibt es auch einige Probleme, die

mehrere rekursive Aufrufe benötigen – in diesem Fall ergibt sich ein Aufrufbaum. Ein bekanntes Beispiel sind die Fibonacci-Zahlen. Die Fibonacci-Zahlen berechnen sich nach der folgenden Formel:

$$\mathsf{fib}(n) = \begin{cases} n & \text{, falls } 0 \leq n \leq 1 \\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2) & \text{, falls } n > 1 \end{cases}$$

Selbsttestaufgabe 4.2.5

Berechnen Sie alle Fibonacci-Zahlen bis $\mathsf{fib}(8)$.

Musterlösung zu Selbsttestaufgabe 4.2.5

$$\mathsf{fib}(0) = 0$$
$$\mathsf{fib}(1) = 1$$
$$\mathsf{fib}(2) = 1$$
$$\mathsf{fib}(3) = 2$$
$$\mathsf{fib}(4) = 3$$
$$\mathsf{fib}(5) = 5$$
$$\mathsf{fib}(6) = 8$$
$$\mathsf{fib}(7) = 13$$
$$\mathsf{fib}(8) = 21$$

Die Implementierung einer passenden Methode ist nach dem gleichen Muster wie oben möglich. Negative Werte für n sind nicht zulässig; bei negativen Werten werfen wir eine ArgumentException.

```
public int fibo(int n)
{
    if (n < 0) throw new ArgumentException();
    if (n < 2) return n;
    return fibo(n - 1) + fibo(n - 2);
}
```

Selbsttestaufgabe 4.2.6

Implementieren Sie eine iterative Lösung für die Berechnung der Fibonacci-Zahlen in der Methode int fibIterative(int n).

Musterlösung zu Selbsttestaufgabe 4.2.6

```
public int fibIterative(int n)
{
    if (n < 0) throw new ArgumentException();
    if (n < 2) return n;
    int x = 0;
    int y = 1;
    for(int i = 2; i <= n; ++i)
    {
        int z = x + y;
        x = y;
        y = z;
    }
    return y;
}
```

Selbsttestaufgabe 4.2.7

In der Programmierung werden bisweilen Zufallszahlen benötigt. Es gibt Formeln zur Erzeugung sogenannter Pseudozufallszahlen, die eine Folge zufälliger Zahlen simulieren. Eine mögliche Formel zur Erzeugung solcher Zahlen ist die folgende:

$$f(n) = \begin{cases} n+1 & \text{, falls } n < 3 \\ 1 + (((f(n-1) - f(n-2)) \cdot f(n-3)) \bmod 100) & \text{, sonst} \end{cases}$$

Implementieren Sie eine Methode int random(int n), die rekursiv $f(n)$ berechnet. Implementieren Sie außerdem eine Methode void printRandomNumbers(), die die Pseudozufallszahlen $f(5)$ bis einschließlich $f(30)$ ausgibt.

Musterlösung zu Selbsttestaufgabe 4.2.7

```
public int random(int n)
{
    if (n < 3) return n+1;
    int r1 = random(n - 1);
```

```
    int r2 = random(n - 2);
    int r3 = random(n - 3);
    return 1 + (((r1 - r2) * r3) % 100);
}

public void printRandomNumbers()
{
    for (int i = 5; i < 31; ++i)
        Console.WriteLine(random(i));
}
```

Bisher haben wir noch nicht weiter darüber nachgedacht, wie es funktionieren kann, dass eine Methode sich selbst aufruft. In C# wird bei einem Methodenaufruf ein Methodenrahmen (StackFrame) erzeugt. In diesem Methodenrahmen sind alle aktuellen Werte der Parameter und lokalen Variablen, die aktuelle Zeile an der sich die Ausführung der Methode gerade befindet [1], sowie ein Verweis auf die aufrufende Methode gespeichert. Die Anweisungen einer Methode existieren nur einmal, sie sind nicht in jedem Methodenrahmen gespeichert. Ein Methodenrahmen kann optisch wie folgt dargestellt werden:

Name
Programmzeile:
Parameter / lokale Variablen:

Ruft nun eine Methode eine andere auf, so wird ein neuer Methodenrahmen erzeugt, und die Parameter und lokalen Variablen werden mit ihren Werten initialisiert. Dabei macht es keinen Unterschied, ob eine Methode eine andere oder eben sich selbst aufruft. Ist die aufgerufene Methode beendet, so wird dies dem Aufrufer – ggf. zusammen mit einem Rückgabewert – mitgeteilt.

Gegeben seien die beiden folgenden Methoden:

```
int a(int x)
{
    int y = 2 * x;
    int z = 3;
    int w = b(y, z) + x;
    return w;
}

int b(int c, int d)
{
    int e = c + 2 * d;
    return e;
}
```

[1] Genauer muss man sich sogar die genaue Anweisung merken, die gerade ausgeführt wird; davon abstrahieren wir an der Stelle.

Beim Aufruf a(3) wird ein Methodenrahmen für den Aufruf a(3) erzeugt:

Name: a
Programmzeile: 0
Parameter / lokale Variablen: x = 3

Durch Ausführung der ersten beiden Zeilen ändert sich der Methodenrahmen wie folgt:

Name: a
Programmzeile: 1
Parameter / lokale Variablen: x = 3 y = 6

Name: a
Programmzeile: 2
Parameter / lokale Variablen: x = 3 y = 6 z = 3

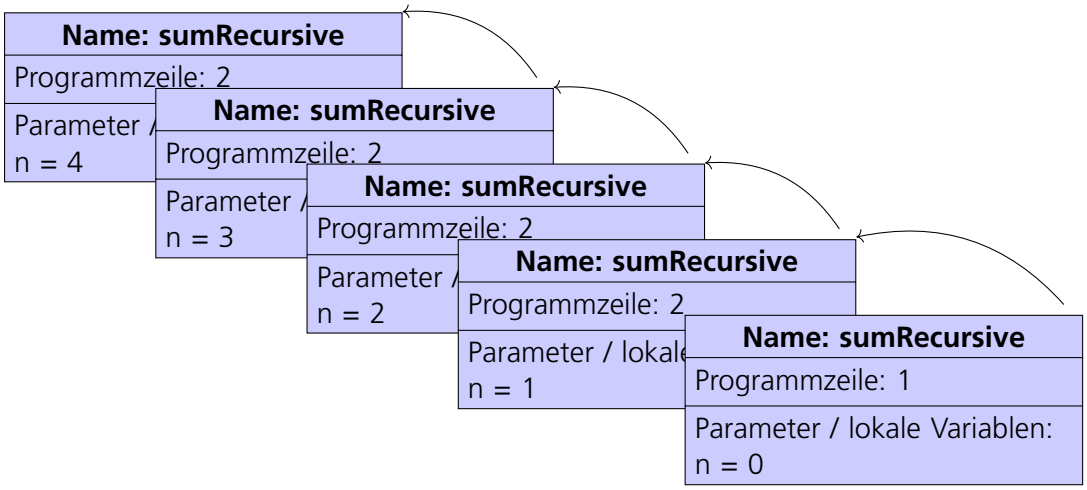
Wenn nun in Programmzeile 3 der Befehl b(6, 3) ausgeführt wird, muss ein neuer Methodenrahmen erzeugt werden. Den Aufrufer kennzeichnen wir über einen Pfeil:

Name: a
Programmzeile: 2
Parameter / lokale Variablen: x = 3 y = 6 z = 3

Name: b
Programmzeile: 0
Parameter / lokale Variablen: c = 6 d = 3

Ist die Ausführung von b(6, 3) beendet, kann der zugehörige Methodenrahmen wieder gelöscht werden und die Ausführung von a(3) wird an der entsprechenden Stelle fortgesetzt.

Bei einer rekursiven Methode passiert das gleiche, nur dass dort die Methodenrahmen alle von der gleichen Methode stammen. Sie werden jeweils mit eigenen Parametern und lokalen Variablen erzeugt. Die folgende Abbildung veranschaulicht die Schritte bei der Ausführung von sumRecursive(4).



Wenn eine Methode eine andere aufruft, so entsteht ein sogenannter Methodenstapel (engl. stack). Vom Methodenstapel wird immer nur die oberste, also die zuletzt aufgerufene Methode ausgeführt. Erst wenn diese beendet ist und wieder vom Stapel entfernt wurde, kann die Methode darunter fortgesetzt werden. Die Datenstruktur des Stapels findet auch in anderen Bereichen Anwendung (mehr dazu in der nächsten Lektion).

Bemerkung 4.2.3 : StackOverflowException

Wenn nicht mehr genug Speicherplatz für neue Methodenrahmen vorhanden ist, erzeugt die Laufzeitumgebung in C# eine StackOverflowException. Dieser Fall tritt auf, wenn eine Rekursion niemals den Basisfall erreicht oder den Basisfall erst nach zu vielen Schritten erreichen würde.

Rekursionen können nicht nur bei mathematischen Funktionen verwendet werden, sondern auch in vielen anderen Bereichen.

Ein Palindrom ist ein Wort, dass sowohl vorwärts als auch rückwärts gelesen das gleiche ist.

Selbsttestaufgabe 4.2.8

Implementieren Sie die Methode bool palindromeIterative (String s), die iterativ prüft ob es sich bei der Zeichenkette um ein Palindrom handelt.

Musterlösung zu Selbsttestaufgabe 4.2.8

```
public bool palindromeIterative(String s)
{
    for (int i = 0; i < s.Length / 2; ++i)
        if (s[i] != s[s.Length - i-1]) return false;
```

```
    return true;
}
```

Der Begriff Palindrom lässt sich auch rekursiv definieren. Ein Wort aus einem oder keinen Zeichen ist immer ein Palindrom. Ein längeres Wort ist dann ein Palindrom, wenn der erste und letzte Buchstabe identisch sind und der Rest der Zeichenkette, also ohne den ersten und letzten Buchstaben, auch ein Palindrom ist.

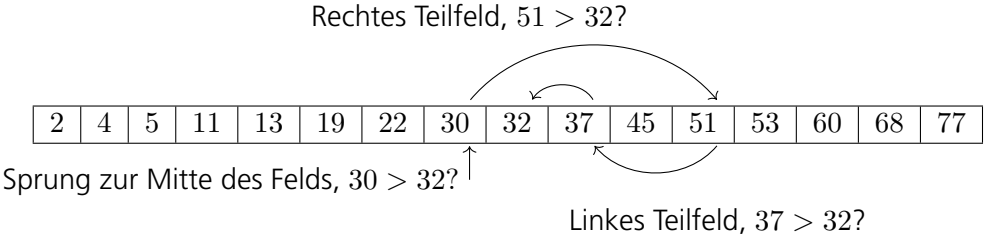
Selbsttestaufgabe 4.2.9

Implementieren Sie die Methode bool palindromeRecursive(String s) , die mit Hilfe der rekursiven Definition prüft, ob es sich bei der Zeichenkette um ein Palindrom handelt.

Musterlösung zu Selbsttestaufgabe 4.2.9

```
public bool palindromeRecursive(String s)
{
    if (s.Length < 2) return true;
    return palindromeRecursive(s.Substring(1, s.Length - 2))
        && s[0] == s[s.Length - 1];
}
```

Auch für das Suchen in und das Sortieren von Feldern gibt es einige rekursive Algorithmen. Bisher haben wir ein sortiertes Feld immer iterativ von einem Ende zum anderen durchsucht. Dieses Verfahren wird auch lineare Suche genannt. Wir können jedoch auch das mittlere Element eines sortierten Feldes betrachten und entscheiden, in welcher der beiden Hälften sich unser gesuchtes Element befindet. Anschließend halbieren wir die Hälfte wieder und treffen die gleiche Entscheidung. Wir gehen solange so vor, bis die zu durchsuchende Menge maximal noch 2 Elemente beinhaltet. Die folgende Grafik illustriert die Suche mit diesem Verfahren nach dem Eintrag 32.



Wir können die Methode bool binarySearch(int value, int[] array, int start, int end), die im Bereich start bis einschließlich end im Feld nach dem Wert sucht, folgendermaßen rekursiv implementieren:

Algorithmus 4.2.4 : Binäre Suche

```
private bool binarySearch(int value, int[] array,
    int start, int end)
{
    //Basisfall: Bereich enthält maximal 2 Elemente
    if (end - start <= 1)
        return array[start] == value || array[end] == value;
    //sonst: rekursive Aufteilung
    //Mitte bestimmen
    int center = (start + end) / 2;
    //Ist Wert in der Mitte?
    if (array[center] == value) return true;
    //Falls nein: Müssen wir im linken Teilfeld weitersuchen?
    if (array[center] > value)
        return binarySearch(value, array, start, center - 1);
    //Falls nein müssen wir im rechten Teilfeld weitersuchen.
    return binarySearch(value, array, center + 1, end);
}
```

Eine Schwäche dieses Ansatzes, insbesondere im Hinblick auf das Implementationsgeheimnis, ist, dass der Aufruf der Methode implizit Informationen über die Funktionsweise der Methode voraussetzt. Um in einem Feld einen bestimmten Wert zu suchen, bedarf es eigentlich keiner zusätzlicher Parameter, die Start- und Ende-Index anzeigen. Es ist daher in vielen Fällen bei der Angabe einer rekursiven Lösung sinnvoll, die eigentliche rekursive Methode als private zu deklarieren und nach außen hin eine public Methode anzubieten, die nur die probleminhärent notwendigen Parameter erfordert. Diese Methode ruft dann die private rekursive Methode mit den geeigneten Startparametern auf. Damit wird die Rekursion verschleiert und wenn man sich später entscheidet, ein anderes, beispielsweise iteratives, Verfahren für das gleiche Problem zu verwenden, sind ausschließlich lokale Änderungen in der Klasse notwendig, in der das rekursive Verfahren implementiert ist. In unserem Fall der binären Suche könnte eine solche nach außen sichtbare Methode wie folgt aussehen:

```
public bool binarySearch(int value, int[] array)
{
    return binarySearch(value, array, 0, array.Length - 1);
}
```

Dieser Algorithmus wird als binäre Suche (engl. binary search) bezeichnet. Natürlich könnte auch schon bei größeren Restmengen auf ein anderes, zum Beispiel iteratives Suchverfahren umgeschaltet werden.

Ein ähnliches Verfahren wenden wir auch häufig an, wenn wir zum Beispiel in einem Le-

xikon nach einem bestimmten Begriff suchen. Wir schlagen eine Seite auf, sehen nach, ob wir uns vor oder nach dem gesuchten Wort im Alphabet befinden, und wählen dann aus, in welchem Teil wir weiter suchen. Dabei lassen wir jedoch bei der Auswahl der nächsten Seite unser Wissen über die Position der Buchstaben im Alphabet mit einfließen, so dass wir nicht immer genau die Mitte des entsprechenden Teils auswählen. Dieses Wissen steht bei der binären Suche nicht zur Verfügung. Wird Wissen über die Verteilung bei der Suche berücksichtigt, so spricht man von einer Interpolationssuche.

Selbsttestaufgabe 4.2.10

Führen Sie auf dem folgenden Array eine binäre und eine lineare Suche nach dem Element 38 aus. Wie viele Vergleiche benötigen Sie, bis Sie das Element gefunden haben?

$$[3, 7, 14, 16, 18, 22, 27, 29, 30, 34, 38, 40, 50]$$

Musterlösung zu Selbsttestaufgabe 4.2.10

Das Array hat 13 Einträge. Erstes Vergleichselement hat also den Index 7 und den Wert 27. $38 > 27$, also suchen wir im rechten Teilfeld weiter:

$$[29, 30, 34, 38, 40, 50]$$

Dieses Feld hat sechs Einträge, also untersuchen wir den Eintrag an Stelle 3, die 34. Es gilt $38 > 34$, also suchen wir im rechten Teilfeld weiter:

$$[38, 40, 50]$$

Dieses Feld hat drei Einträge, also untersuchen wir den Eintrag an der Stelle 2, die 40. Es gilt $38 > 40$, also suchen wir im linken Teilfeld weiter:

$$[38]$$

Dieses Feld hat nur einen Eintrag und es gilt $38 = 38$, also haben wir den gesuchten Eintrag gefunden. Insgesamt haben wir vier Vergleiche durchgeführt. Bei der linearen Suche wären hingegen elf Vergleiche notwendig gewesen.

Selbsttestaufgabe 4.2.11

Schreiben Sie eine Methode `bool binarySearch(String s, String[] feld)`, die mithilfe der binären Suche prüft, ob die Zeichenkette `s` in dem Feld `feld` enthalten ist. Wir gehen dabei davon aus, dass `feld` gemäß der `String.Compare`-Methode von C# sortiert ist,

das heißt wenn String.Compare(s, t) = 1, dann steht s hinter t in feld. Sie dürfen (und sollten) eine private Hilfsfunktion mit zusätzlichen Parametern nutzen.

Musterlösung zu Selbsttestaufgabe 4.2.11

```
public bool binarySearch(String value, String[] array)
{
    return binarySearch(value, array, 0, array.Length - 1);
}
```

```
private bool binarySearch(String value, String[] array
        , int start, int end)
{
    if (end - start <= 1)
        return array[start] == value || array[end] == value;
    int center = (start + end) / 2;
    if (array[center] == value) return true;
    if (String.Compare(array[center], value)==1)
        return binarySearch(value, array, start, center-1);
    return binarySearch(value, array, center + 1, end);
}
```

Wir werden nun zwei Sortierverfahren kennenlernen, die aus einer rekursiven Charakterisierung hervorgehen und bei großen zu durchsuchenden Feldern schneller terminieren als die grundlegenden Suchalgorithmen, die wir bereits besprochen haben. Wir beginnen mit der Betrachtung des Quicksort-Algorithmus. Bei diesem Verfahren wird nach einem beliebig gewählten Verfahren ein Element des Feldes als sogenanntes Pivotelement ausgewählt. Anschließend werden die Elemente des Feldes aufgeteilt. In dem einen Teil befinden sich alle Elemente, die kleiner als das Pivotelement sind und im anderen alle, die größer als das Pivotelement sind. Anschließend werden beide Teile wiederum mit dem gleichen Verfahren aufgeteilt. Bei Teilen mit maximal zwei Elementen kann die Sortierung dann direkt vorgenommen werden. Anschließend ist das gesamte Feld sortiert.

Algorithmus 4.2.5 : Quicksort (abstrakt)

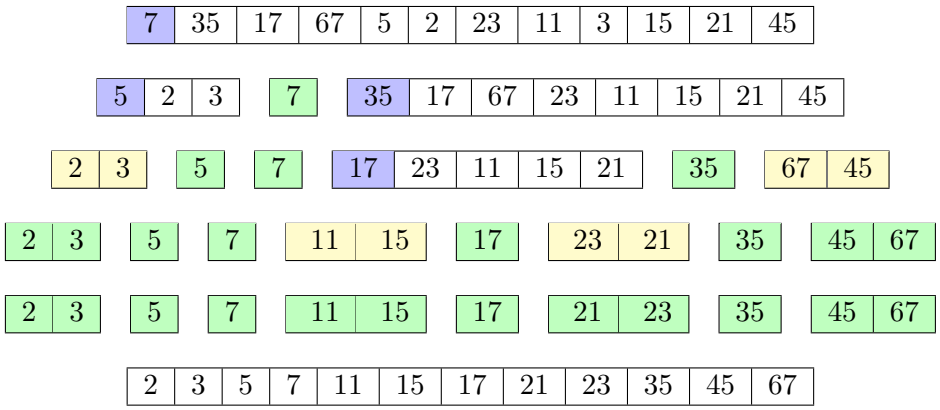
Gegeben: Ein Feld array, das nicht notwendigweise sortiert ist
Gesucht: Ein Feld sorted, das die gleichen Einträge hat wie array, das aber zudem sortiert ist.

- Wähle einen beliebigen Eintrag p aus array. Wir nennen p das Pivotelement.
- Erstelle zwei Felder a und b. Das Feld a enthalte alle Elemente von array, die

kleiner als p sind und b enthalte alle Elemente von array, die größer als p sind.

- Sortiere a und b mit Quicksort.
- Setze sorted $=$ a $+ [p] +$ b, wobei wir das Feld, das durch das Hintereinander anfügen zweier Felder x und y entsteht als $x + y$ bezeichnen.

Die folgende grafische Darstellung veranschaulicht dieses Verfahren. Wir wählen hierbei als Pivotelement immer das erste Element des Feldes und markieren das Pivotelement blau. Fertig sortierte Teilfelder markieren wir grün und Teilfelder, die durch direkten Vergleich sortiert werden können gelb.



Um dieses Verfahren zu implementieren teilen wir den Algorithmus in zwei Teile. Das Aufteilen des Feldes implementieren wir in der Methode split und das Sortieren in der Methode quicksort. Wir beachten insbesondere, dass das Verfahren, wie wir es bisher besprochen haben an zwei Stellen unterspezifiziert ist:

- Wir können ein beliebiges Element als Pivotelement wählen. Von Hand haben wir hier immer das erste Element verwendet, man könnte aber ebenso beispielsweise ein zufälliges Element oder das letzte Element als Pivotelement verwenden.
- Wie wir die Elemente in den Teilfelder anordnen, die durch die Pivotisierung entstehene, ist nicht spezifiziert. Von Hand haben wir einfach die Reihenfolge in den Teilfeldern beibehalten, grundsätzlich darf die Ordnung der Teilfeldern beim Aufteilen aber beliebig sein.

Wir werden weiterhin auch in der Implementierung des Algorithmus das erste Element des Feldes als Pivotelement verwenden. Bei der Reihenfolge der Elemente in den Teilfeldern werden wir allerdings aus Effizenzgründen einen anderen Weg wählen. In der Darstellung die wir gewählt haben, in der wir beim Aufteilen tatsächlich neue Teilfelder erstellen, ist zwar instruktiv, aber nicht effizient, weil die Bereitstellung des Speicherplatzes und das löschen der Felder aus den Zwischenschritten einen großen Zusatzaufwand bedeutet. Stattdessen arbeiten wir in dem ursprünglichen Feld und sortieren die Einträge in dem Feld um, um die gewünschten Teilfelder zu erhalten. Hierbei bleibt die vorherige Ordnung nicht erhalten.

Wir geben zunächst die Implementation von quicksort an, die auf eine Implementation der split-Methode angewiesen ist, die wir erst im Anschluss angeben. Wir geben nur die private rekursive Methode an.

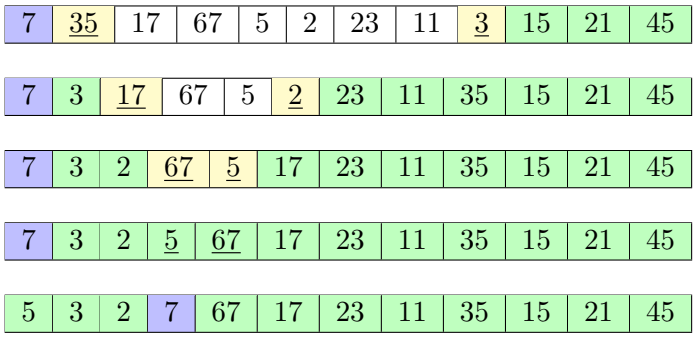
Algorithmus 4.2.6 : Quicksort in C#

```
private void quicksort(int[] array, int start, int end)
{
    //Basisfall 1: Maximal 1 Element (schon sortiert):
    if (end <= start) return;
    //Basisfall 2: 2 Elemente (direkt sortieren):
    if (end == start + 1)
    {
        if (array[start]>array[end]) swap(array, start, end);
        return;
    }
    //Feld aufteilen
    int pivot = split(array, start, end);
    //linken Teil (ohne Pivot) sortieren
    quicksort(array, start, pivot - 1);
    //rechten Teil (ohne Pivot) sortieren
    quicksort(array, pivot + 1, end);
}
```

Hier und im Folgenden verwenden wir die Methode swap als Hilfsfunktion:

```
public void swap(int[] array, int index1, int index2)
{
    int temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
}
```

Um das Feld entsprechend aufzuteilen, wählen wir das erste Element als Pivotelement. Anschließend beginnen wir, vom zweiten Element an aufsteigend, ein Element zu suchen, das größer als das Pivotelement ist. Ebenso beginnen wir, vom letzten Element an absteigend, ein Element zu suchen, das kleiner ist als das Pivotelement. Haben wir diese beiden Elemente gefunden, so vertauschen wir sie und suchen von den Positionen aus weiter. Das Vertauschen endet, sobald sich die Suchindizes von links und rechts treffen. Das Element an der Grenze wird anschließend mit dem Pivotelement vertauscht. Dieses Vorgehen ist in der folgenden Abbildung dargestellt. Das Pivotelement ist wieder blau markiert, die Elemente, die bereits untersucht sind und in dem richtigen Teilfeld stehen grün und die aktuellen Vergleichsindizes gelb.



In C# ergibt sich folgende Implementation des Verfahrens:

Algorithmus 4.2.7 : split **für** quicksort **in C#**

```
public int split(int[] array, int start, int end)
{
    //Linkes Ende des zu zerteilenden Feldes
    int l = start + 1;
    //Rechtes Ende des zu zerteilenden Feldes
    int r = end;
    //Das erste Element des übergebenen Feldes ist Pivot
    int pivot = array[start];
    //laufe mit dem linken Zeiger nach links
    //und dem rechten nach recht, bis sie
    //aufeinandertreffen.
    while (l < r)
    {
        while (array[l] <= pivot && l < r) ++l;
        while (array[r] > pivot && l < r) --r;
        //Wenn links ein zu großes und rechts ein
        //zu kleines Element gefunden wurde, tausche sie
        swap(array, l, r);
    }
    //Falls das letzte zu große Element links keinen rechten
    //Gegenpart mehr gefunden hat, muss das letzte Element
    //links in den rechten Array rutschen.
    if (array[l] > pivot) --l;
    //Tausche das Pivotelement mit letztem linken Element.
    //Dann steht das Pivotelement genau zwischen den
    //kleineren Elementen auf der linken und den größeren
    //Elementen auf der rechten Seite.
    swap(array, l, start);
    //Gib Index des Pivotelements zurück.
```

```
    return l;
}
```

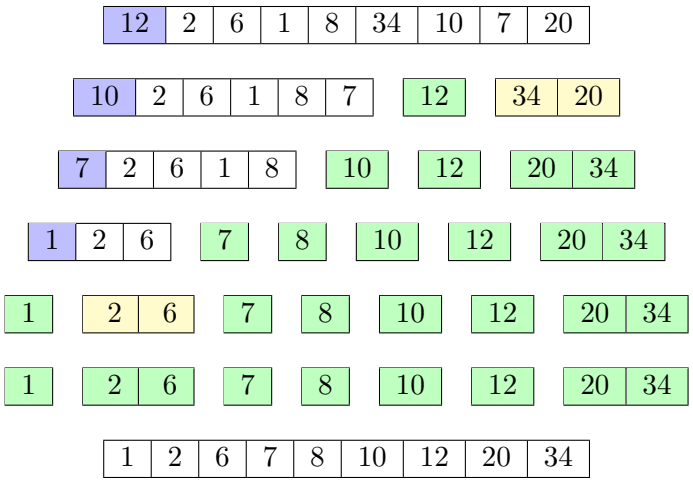
Selbsttestaufgabe 4.2.12

Führen Sie für das folgende Feld Schritt für Schritt Quicksort aus, wie es in der C#-Implementation umgesetzt ist:

$$[12, 2, 6, 1, 8, 34, 10, 7, 20]$$

Beachten Sie insbesondere die Aufteilung gemäß split.

Musterlösung zu Selbsttestaufgabe 4.2.12



Ein weiteres rekursives Sortierverfahren ist das „Sortieren durch Verschmelzen“ (Merge Sort). Bei diesem Verfahren wird im Gegensatz zu Quicksort nicht beim Aufteilen sortiert, sondern erst wieder beim Zusammenfügen. Das Feld wird in zwei möglichst gleich große Teile aufgeteilt, die nach dem gleichen Verfahren sortiert werden. Die beiden sortierten Teilfelder werden nun zu einer sortierten Gesamtliste vereint, indem die beiden ersten Elemente verglichen und das kleinere aus seinem Teil entnommen und in das Zielfeld übernommen wird. Das wird solange fortgesetzt, bis beide Teilfelder leer sind. Die Verschmelzungsoperation lässt sich wie folgt definieren:

Definition 4.2.8 : Verschmelzungsoperation merge

Gegeben seien zwei sortierte Felder $A = [a_1, ..., a_n]$ und $B = [b_1, ..., b_m]$, dann erzeugen wir ein sortiertes Feld C, das alle Einträge von A und B beinhaltet rekursiv wie

folgt:

$$\text{merge}(A,B) = \begin{cases} A & \text{falls } B \text{ leer ist} \\ B & \text{falls } A \text{ leer ist} \\ [a_1] + \text{merge}([a_2,...,a_n],B) & \text{falls } A,B \text{ nicht leer sind und } a_1 \leq b_1 \\ [b_1] + \text{merge}(A,[b_2,...,b_m]) & \text{sonst} \end{cases}$$

Wir können merge zum Beispiel wie folgt implementieren:

Algorithmus 4.2.9 : Implementation von merge

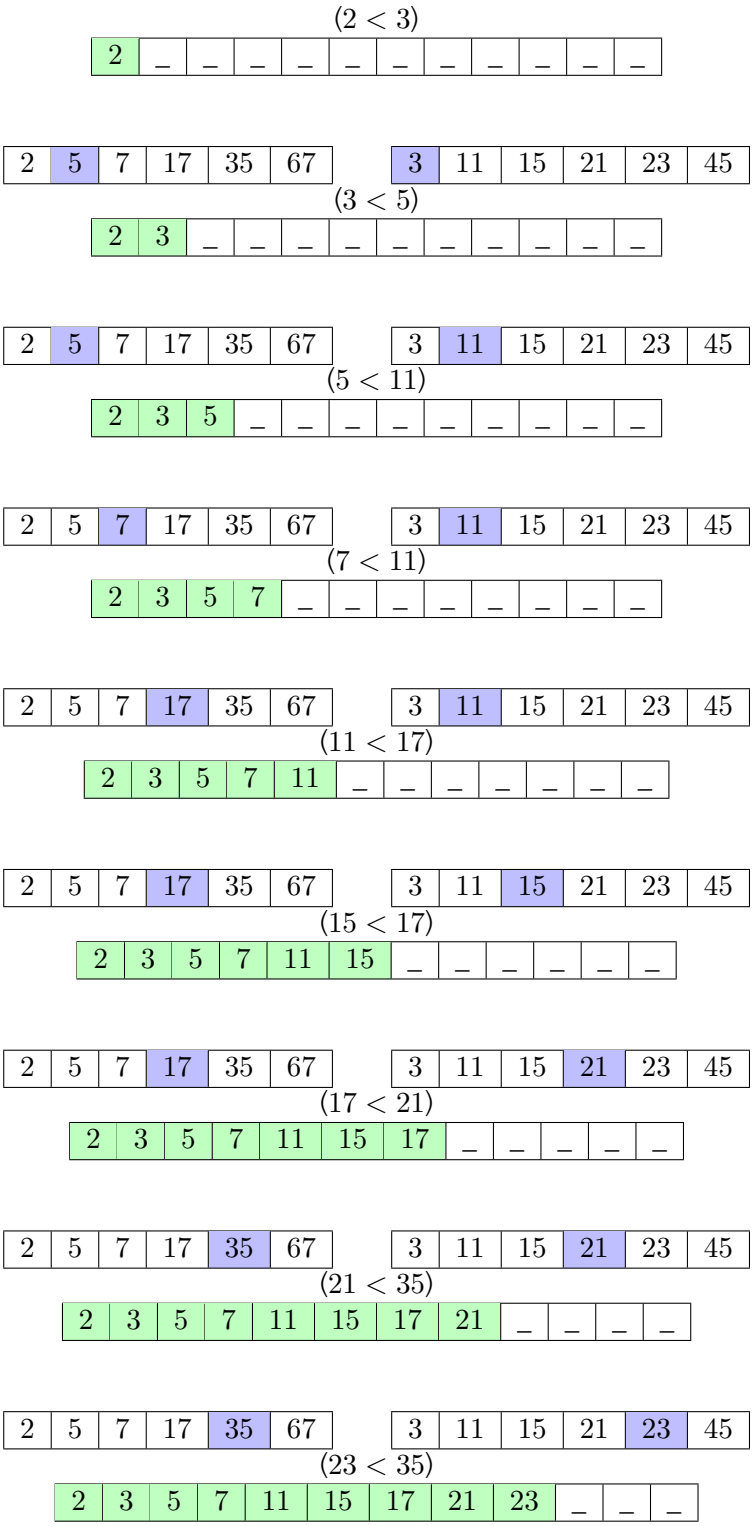
```
int[] merge(int[] A, int[] B)
{
    int[] C = new int[A.Length + B.Length];
    //Durchlaufindizes für A, B und C
    int a = 0;
    int b = 0;
    int c = 0;
    //Fülle C bis alle Einträge von A und B kopiert wurden.
    while(a < A.Length || b < B.Length)
    {
        //Wurden bereits alle Einträge von B kopiert,
        //nimm den ersten Eintrag von A
        if (b == B.Length) C[c++] = A[a++];
        //Wurden bereits alle Einträge von A kopiert,
        //nimm den ersten Eintrag von B
        else if (a == A.Length) C[c++] = B[b++];
        //Sonst nimm den kleineren der beiden ersten
        //Einträge von A und B.
        else if (A[a] < B[b]) C[c++] = A[a++];
        else C[c++] = B[b++];
    }
    return C;
}
```

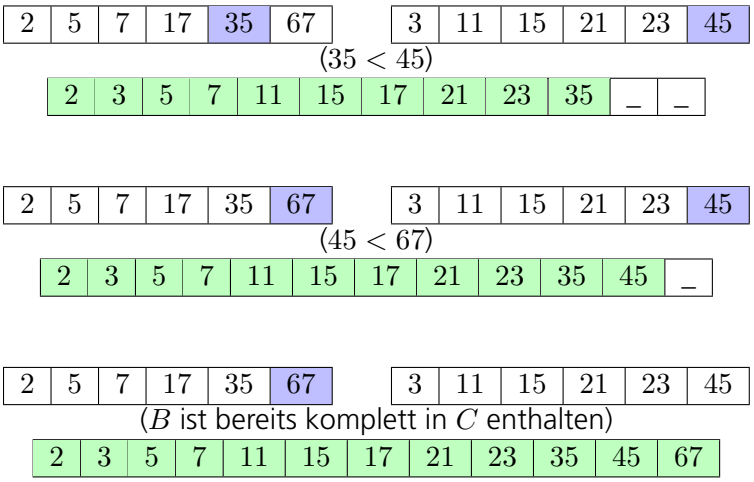
Alternativ könnte man merge auch direkt aus der rekursiven Definition heraus implementieren, in diesem Fall würde das aber die Erstellung vieler unnötiger Felder bedürfen oder aber die Lesbarkeit verringern.

Die folgende Darstellung illustriert das merge-Verfahren für zwei Beispielsfelder $A = [2,5,7,17,35,67]$ und $B = [3,11,15,21,23,45]$; das Ergebnisfeld nennen wir C:

2	5	7	17	35	67

3	11	15	21	23	45


(2 < 3)
2 _ _ _ _ _ _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(3 < 5)
2 3 _ _ _ _ _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(5 < 11)
2 3 5 _ _ _ _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(7 < 11)
2 3 5 7 _ _ _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(11 < 17)
2 3 5 7 11 _ _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(15 < 17)
2 3 5 7 11 15 _ _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(17 < 21)
2 3 5 7 11 15 17 _ _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(21 < 35)
2 3 5 7 11 15 17 21 _ _ _ _
2 5 7 17 35 67
3 11 15 21 23 45
(23 < 35)
2 3 5 7 11 15 17 21 23 _ _ _



Eine Implementation von Merge Sort auf Feldern könnte dann so aussehen:

Algorithmus 4.2.10 : Merge Sort auf Feldern

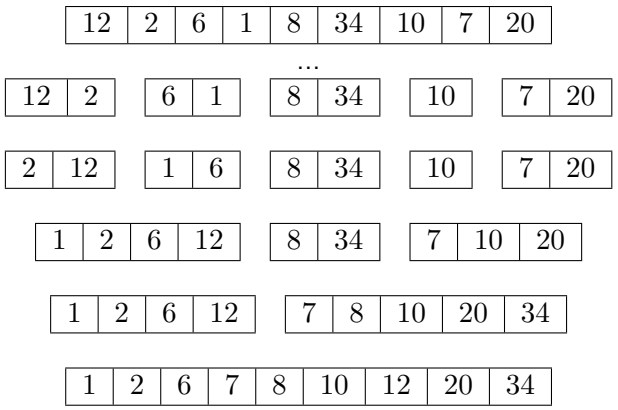
```
public int[] mergeSort(int[]a)
{
    //Einzelne Elemente und leere Arrays sind sortiert.
    if (a.Length <= 1) return a;
    //Teile das Feld in der Mitte
    int center = a.Length / 2;
    int[] l = new int[center];
    int[] r = new int[a.Length - l.Length];
    int i = 0;
    while (i < l.Length)
        l[i] = a[i++];
    while (i < a.Length)
        r[i - l.Length] = a[i++];
    //Sortiere die Teilfelder rekursiv und verschmelze
    //die sortierten Teilfelder mit merge.
    return merge(mergeSort(l), mergeSort(r));
}
```

Selbsttestaufgabe 4.2.13

Sortieren Sie das folgende Feld mit Merge Sort:

$$[12, 2, 6, 1, 8, 34, 10, 7, 20]$$

Musterlösung zu Selbsttestaufgabe 4.2.13



Quicksort und Merge Sort sind sicherlich auf den ersten Blick weniger intuitiv als die zuvor besprochenen Sortierverfahren wie Insertion Sort oder Bubblesort. Dass diese Verfahren dennoch in der Praxis Anwendung finden, kann man anhand der folgenden Überlegung nachvollziehen:

Wie lange benötigt Sortieren beim Einfügen um eine Array der Länge n zu sortieren? Das hängt natürlich stark von der Eingabe ab. Der ungünstigste Fall für dieses Verfahren ist gegeben, wenn man ein Feld von n Zahlen sortieren möchte, die absteigend sortiert sind. In diesem Fall müssen wir bei jeder Einfügeoperation die komplette Ergebnisliste durchlaufen. Wir benötigen also

$$0+1+2+3+...+n-1=\sum_{i=0}^{n-1} i=\sum_{i=1}^{n} i-n=(n+1)\cdot\frac{n}{2}-n=\frac{n^2}{2}+\frac{n}{2}-n=\frac{n^2}{2}-\frac{n}{2}$$

Vergleichsschritte. Der dominante Term, der hier das Laufzeitverhalten charakteristisch beschreibt[2] ist n^2. Die selbe Analyse und das selbe Beispiel zeigen das gleiche Laufzeitverhalten im ungünstigsten Fall für Bubblesort. Man kann außerdem zeigen, dass auch im durchschnittlichen Fall die Zahl der Vergleichsoperationen in der Größenordnung von n^2 liegt. Im günstigsten Fall, dem vollständig sortierten Array, hingegen werden nur n Vergleichsoperationen benötigt.

Merge Sort hingegen benötigt, unabhängig von der Eingabe, in jeder Rekursionsebene n Vergleichsoperationen, um die Verschmelzung durchzuführen. Da die Unterteilung wahllos in der Hälfte des Feldes erfolgt, ist hierzu gar keine Vergleichsoperation notwendig. Insgesamt gibt es $\lceil \log_2(n) \rceil$ Ebenen, die Zahl der Vergleichsoperationen von Merge Sort ist also (im günstigsten, ungünstigsten und mittleren) Fall stets $\log_2 n \cdot n$. Im Durchschnitt und im schlimmsten Fall ist Merge Sort also (für große zu sortierende Felder) signifikant

[2] Der Grund hierfür ist, dass mit steigendem n Der Teilterm n^2 den Teilterm n dominiert. Konstante Vorfaktoren wie hier $\frac{1}{2}$ werden bei Laufzeitanalysen zwecks Ausbildung sprechender Klassen zudem üblicherweise ignoriert. Wir erinnern an die Landau-Notation, die es erlaubt die hier errechnete Zahl an Vergleichsoperationen als $O(n^2)$ zu schreiben.

schneller als die einfacheren Verfahren. Man beachte aber, dass Merge Sort keinen Vorteil daraus ziehen kann, wenn ein Array bereits (beinahe) sortiert ist.

Quicksort hingegen liegt in der gleichen Größenordnung wie Merge Sort was die Zahl der Vergleichsoperationen im mittleren Fall und im besten Fall anbelangt; im schlechtesten Fall – der durch ein vollständig sortiertes Array erreicht werden kann – kann Quicksort allerdings mit Sortieren durch Verschmelzen nicht mithalten und benötigt in der Größenordnung von n^2 Vergleichsoperationen. Dadurch, dass der Verwaltungsaufwand bei Quicksort wesentlich geringer ist als bei Sortieren durch Verschmelzen, wird in der Praxis dennoch oft Quicksort statt Sortieren durch Verschmelzen verwendet. Insgesamt haben die verschiedenen Sortierverfahren verschiedene Vorteile, so dass es sich lohnt, Bubblesort, Quicksort und Merge Sort als Implementation bereitzuhalten:

- Für nahezu sortierte oder kurze Arrays lohnt sich Bubblesort, da es den geringsten Verwaltungsaufwand hat und nahezu sortierte Felder schnell sortiert.

- Bei größeren Arrays unbekannter Sortierung ist Quicksort der Algorithmus der Wahl, da es eine niedrige durchschnittliche Zahl an Vergleichsoperationen mit einem moderaten Verwaltungsaufwand kombiniert.

- Bei sehr großen Feldern schließlich fallen die Nachteile des höheren Verwaltungsaufwands von Merge Sort nicht mehr ins Gewicht und die Zahl der Vergleichsoperationen dominiert das Laufzeitverhalten. Dann lohnt es sich, zur Vermeidung der ungünstigen Fälle bei Quicksort, Sortieren durch Verschmelzen zu verwenden.

4.3 Dynamische Programmierung

Bei der Entwicklung rekursiver Lösungen haben wir bislang nicht berücksichtigt, ob wir gegebenenfalls durch den rekursiven Abstieg Teillösungen mehr als ein Mal berechnen. Allerdings ist das keine unerhebliche Fragestellung, denn der Rechenaufwand, der durch wiederholtes Berechnen der gleichen Teillösungen anfällt, kann signifikant sein.

Betrachten wir beispielsweise unser Programm zur Berechnung der Fibonacci-Zahlen, am Beispiel der Eingabe 5. Um die fünfte Fibonacci-Zahl zu berechnen, benötigen wir die vierte und die dritte Fibonacci-Zahl. Um die vierte Fibonacci-Zahl zu berechnen, benötigen wir wiederum die dritte und die zweite Fibonacci-Zahl. Die folgende Grafik stellt den entsprechenden Aufrufbaum dar, wir schreiben zur einfacheren Darstellung $f(\_)$ für fib$(_)$:

Lektion 6: Grenzen der Algorithmik: Berechenbarkeit und Komplexität

Wir haben in den vorangegangenen Lektionen zahlreiche Algorithmen kennengelernt, die es uns ermöglichen, eine Vielzahl von Fragestellungen automatisch zu beantworten. Doch lässt sich jedes wohldefinierte Problem mit einem Computerprogramm lösen? Und wenn ein Problem grundsätzlich lösbar ist, ist es dann auch effizient lösbar? Diesen Fragen möchten wir in dieser Lektion nachgehen. Zu diesem Zweck betrachten wir eine Teilmenge von C#, die für Entscheidungsprobleme verwendet werden kann. Zweck dieser Einschränkung ist, dass wir uns in Beweisen auf einen überschaubaren Satz an Befehlen zurückziehen können. Hinsichtlich der Ausdrucksstärke ändert sich dadurch im Vergleich zu dem vollen Befehlssatz nichts.

Definition 6.1 : Decision-C#

Decision-C# ist eine Teilmenge von C#, die die folgenden Sprachkonstrukte enthält:

- Variablen vom Typ `int` oder `bool`. Wir nehmen für die Zwecke dieser Lektion an, dass der Datentyp `int` alle ganzen Zahlen fassen kann und der Platzverbrauch einer `int`-Variable `n` von ihrem Wert abhängig ist und $1 + log_2\lceil \texttt{n} \rceil$ beträgt.
- Die Konstanten `true`, `false`, sowie alle ganzen Zahlen $\mathbb{Z}$
- Das Trennsymbol ; und die Blockbegrenzer $\{, \}$
- Die Operatoren $=, ==, +, -, *, /, <, >$
- Die Schlüsselwörter `while`, `if`, `else`, `int`, `bool`
- *Entscheidungsmethoden*, das heißt Methoden, die eine beliebige Zahl an `int`-Parametern erhalten und einen `bool`-Wert zurückgeben.
- *Berechnungsmethoden*, das heißt Methoden, die eine beliebige Zahl an `int`-Parametern erhalten und einen `int`-Wert zurückgeben.

Die Einschränkungen an Decision-C# im Vergleich zu C# sind nicht substanziell. Der Rückgriff auf das von Neumann-Modell macht deutlich, dass mit diesem reduzierten Befehlssatz alle Funktionalitäten von C# umgesetzt werden können. Umgekehrt ist die spezielle Interpretation des Datentyps `int` als Datentyp für alle ganzen Zahlen sehr wohl eine ent-

scheidende Änderung. Wir nennen Entscheidungsmethoden und Berechnungsmethoden einfach Programme, wenn im Kontext klar ist, ob wir von einer Entscheidungs- oder einer Berechnungsmethode sprechen.

Konzeptuell entspricht int dem Typ BigInteger, der in C# implementiert ist, allerdings kann ein echter PC natürlich nicht tatsächlich beliebig große Zahlen darstellen, da der Speicher, der dem PC zur Verfügung steht, endlich ist. Da wir uns in dieser Lektion mit den Grenzen der Algorithmik befassen möchten, ist es sinnvoll, von dieser – von PC zu PC individuellen und sich stets verschiebenden – Einschränkung zu abstrahieren. An einigen Stellen, an denen diese Einschränkung für die Stärke oder Gültigkeit von Aussagen eine Rolle spielt, werden wir hierauf noch einmal zu sprechen kommen.

Zwei zentrale Begriffe für die Berechenbarkeitstheorie sind „Berechenbarkeit“ und „Entscheidbarkeit“, die ihre Entsprechung in den Methodentypen finden, die wir für Decision-C# definiert haben, den Entscheidungsmethoden und Berechnungsmethoden.

Definition 6.2 : Berechenbarkeit

Wir nennen eine (partielle) Funktion $f\colon \mathbb{Z}^n \to \mathbb{Z}$, $n \in \mathbb{N}_0$ berechenbar, falls es in Decision-C# eine Berechnungsmethode f gibt, so dass gilt:

$f(x_1, ..., x_n) = y \Leftrightarrow$ der Aufruf f$(x_1, ..., x_n)$ gibt den Wert y zurück.

Diese Definition enthält implizit auch: Wenn $f(x_1, ..., x_n)$ nicht definiert ist, dann terminiert f$(x_1, ..., x_n)$ nicht.

Den Begriff der Berechenbarkeit verdeutlichen wir uns anhand einiger einfacher Beispiele. Tatsächlich werden Sie mit Sicherheit schon einige Berechnungsmethoden implementiert haben.

Beispiel 6.3

Wir betrachten die Funktion

$$f\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, f(x, y) = x^{|y|}$$

Diese Funktion ist berechenbar, beispielsweise können wir die folgende Berechnungsmethode angeben, die f berechnet:

```
int f(int x, int y)
{
    if(y<0) y = -y;
    int z = 1;
    while(y>0)
    {
        z = z * x;
        y = y - 1;
    }
```

```
    return z;
}
```

Ein weiteres Beispiel ist die Funktion

$$max\colon \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}, max(x,y)=\begin{cases}x & \text{, falls } x\geq y\\ y & \text{, sonst}\end{cases}$$

Diese Funktion ist berechenbar, beispielsweise können wir die folgende Berechnungsmethode angeben, die *max* berechnet:

```
int max(int x, int y)
{
    if(x<y) return y;
    return x;
}
```

Um Entscheidungsprobleme zu definieren, müssen wir zunächst an Beispiel 1.2.16 und die Codierungstheorie aus dem Kurs „Einführung in die Informatik 1“ erinnern. Gegeben eine beliebige endliche Menge Σ (das Alphabet), so kann man jede endliche Sequenz (Wort) $w=a_1a_2...a_n\in\Sigma^*$ als n-stellige Zahl zur Basis $|\Sigma|+1$ interpretieren. Wir schreiben $code_\Sigma(w)$ um die Codierung von w als natürliche Zahl zu bezeichnen und $decode_\Sigma(n)$ um die Zeichenkette über Σ zu bezeichnen, die durch die natürliche Zahl n codiert wird. Wir erweitern $decode_\Sigma$ auf alle ganzen Zahlen, indem wir $decode_\Sigma(z)=\epsilon$ setzen für alle $z\in\mathbb{N}_0$ die nicht im Bild von $code_\Sigma$ liegen. Dabei bezeichnet hier und im Folgenden ϵ das leere Wort, als eine Sequenz von null Zeichen aus dem Alphabet Σ. Wir sammeln an dieser Stelle die Grundbegriffe der formalen Sprachen noch einmal kompakt, dabei ist Σ eine endliche Menge:

Σ	Alphabet
$a\in\Sigma$	Alphabetssymbol / -zeichen
Σ^*	Menge der Wörter über Σ
$L\subseteq\Sigma^*$	Sprache über Σ
$w\in L$	Wort w aus der Sprache L
ϵ	leeres Wort (Wort der Länge 0)
$\overline{L}=\Sigma^*\setminus L$	Komplementsprache von L

Beispiel 6.4 : Codierung von Wörtern

Wir betrachten das Alphabet $\Sigma=\{a,b,c\}$ und wollen das Wort $abbc$ als natürliche Zahl codieren. Wir stellen fest, dass $|\Sigma|=3$, also benötigen wir drei Ziffern, um die drei Zeichen a,b,c zu codieren. Eine zusätzliche Ziffer benötigen wir, um Leerzeichen zu codieren, denn angenommen, wir würden beispielsweise das Zeichen a mit 0 codieren, dann wären die Worte a, aa und aaa nicht unterscheidbar – alle drei würden

mit der Zahl 0 codiert werden, denn die Zeichenfolgen 0, 00 und 000 bezeichnen alle die natürliche Zahl 0. Wir wählen $code_\Sigma(a) = 1, code_\Sigma(b) = 2, code_\Sigma(c) = 3$ und erhalten so $code_\Sigma(abbc) = (1223)_4 = 3 \cdot 4^0 + 2 \cdot 4^1 + 2 \cdot 4^2 + 1 \cdot 4^2 = 107$.

Es sei an dieser Stelle gleich angemerkt, dass diese Codierungsform, bei der jedes Symbol mit einer Ziffer repräsentiert wird, nicht die einzige mögliche und auch nicht unbedingt die knappste Repräsentationsform ist. Eine häufig verwendete Alternative ist die Längenlexikografische Aufzählung. Das heißt, dass die Zahl 0 für das leere Wort steht und, die Zahlen 1 bis $|\Sigma|$ für die Wörter der Länge 1 und die Zahlen zwischen $|\Sigma| + 1$ und $|\Sigma| + 1 + |\Sigma|^2$ für die Wörter der Länge 2 – analog wird die Aufzählung für längere Wörter fortgesetzt. Welche Codierung man konkret verwendet, ist an dieser Stelle unerheblich, wichtig ist nur zu sehen, dass es möglich ist, Wörter als natürliche Zahlen zu codieren.

Definition 6.5 : Entscheidungsprobleme und Entscheidbarkeit

Es sei Σ eine endliche Menge und $P \subseteq \Sigma^*$ eine Sprache über Σ. Dann ist das durch P definierte Entscheidungsproblem die Frage, ob ein beliebiges Wort $w \in \Sigma^*$ in P liegt. Wir nennen (w, P) eine Instanz des Problems P.

P ist entscheidbar, wenn es eine Entscheidungsmethode f gibt, so dass gilt:

- Für alle Wörter $w \in \Sigma^*$ gibt $f(code_\Sigma(w))$ (nach endlich vielen Schritten) einen Rückgabewert aus.
- $f(code_\Sigma(w)) =$ true genau dann, wenn $w \in P$.

Um den Begriff der Entscheidbarkeit etwas besser zu verstehen, betrachten wir ein Beispiel für ein entscheidbares Problem:

Beispiel 6.6

Wir betrachten das Problem

$$\text{SQUAREROOT} = \{w \in \Sigma^* \mid \exists n \in \mathbb{N} : code_\Sigma(w) = n^2\}$$

und zeigen, dass es entscheidbar ist.

```
bool sqrt(int x)
{
    for(int i = 1; i<x; i = i + 1)
    {
        if(i*i == x) return true;
    }
    return false;
}
```

Die beiden Begriffe Entscheidbarkeit und Berechenbarkeit hängen inhaltlich eng zusammen, vermöge der charakteristischen Funktion χ_P eines Problems P. P ist entscheidbar genau dann wenn die charakteristische Funktion

$$\chi_P \colon \mathbb{Z} \to \mathbb{Z}, \chi_P(z) = \begin{cases} 1 & \text{, falls } \mathit{decode}_\Sigma(z) \in P \\ 0 & \text{, sonst} \end{cases}$$

berechenbar ist.

Das lässt sich einfach wie folgt einsehen: Angenommen die Berechnungsmethode chi berechne χ_P, dann kann eine Entscheidungsmethode für P wie folgt programmiert werden:

```
bool p(int x)
{
    return chi(x) == 1;
}
```

Selbsttestaufgabe 6.1

Zeigen Sie, dass das Problem

$$\text{PRIM} = \{w \in \Sigma^* \mid code_\Sigma(w) \text{ ist eine Primzahl}\}$$

entscheidbar ist, indem Sie eine Decision-C#-Entscheidungsmethode angeben, die PRIM entscheidet.

Musterlösung zu Selbsttestaufgabe 6.1

Da wir mit der ganzzahligen Division arbeiten, kann dieses Problem beispielsweise durch die folgende Decision-C#-Entscheidungsmethode entschieden werden:

```
bool prim(int x)
{
    for(int i = 2; i<x; i = i + 1)
    {
        if(x/i*i == x) return false;
    }
    return true;
}
```

Bemerkung 6.7

In manchen Fällen ist es übersichtlicher, ein Entscheidungsproblem über Paaren (oder

allgemeiner Tupeln) von Wörtern zu formulieren. In diesen Fällen verwenden wir auch Entscheidungsmethoden mit entsprechend vielen Parametern um sie zu entscheiden. Es ist möglich, Paare von Wörtern $(w, v) \in \Sigma_1^* \times \Sigma_2^*$ auf einfache Weise über dem Alphabet $\Sigma_1 \cup \Sigma_2 \cup \{\square\}$ mit $\square \notin \Sigma_1 \cup \Sigma_2$ als einzelnes Wort $w\square v$ aufzufassen.

Gelegentlich tritt an die Stelle eines Alphabets Σ auch eine ganze oder natürliche Zahl oder eine Struktur wie ein Graph. Ganze und natürliche Zahlen haben bereits die richtige Form um als solche an eine Entscheidungsmethode übergeben zu werden. Wenn Zahlen in Entscheidungsproblemen mit mehreren Parametern vorkommen, kann man sie als Worte über $\{0, 1\}$ auffassen und so das Problem über mehrere Parameter auf einen einzelnen Parameter zurückführen. Graphen können über eine der Repräsentationsformen aus der fünften Lektion, am einfachsten über die Adjazenzmatrix, in Zeichenketten überführt werden.

6.1 Entscheidbarkeit

6.1.1 Das Spezielle Halteproblem

Nachdem wir ein Berechnungsmodell gefunden haben und ein Verständnis dafür gewonnen haben, was es bedeutet, dass ein Problem entscheidbar ist, wollen wir nun Beispiele für Probleme kennen lernen, die nicht entscheidbar sind und eine Beweistechnik erarbeiten, um weitere Probleme als unentscheidbar nachzuweisen. Zunächst betrachten wir hierzu das Halteproblem in verschiedenen Varianten. Bei den verschiedenen Varianten des Halteproblems geht es stets darum, herauszufinden, ob eine Methode mit einer gewissen Eingabe irgendwann anhält, das heißt, ein return-Statement erreicht, oder nicht.

Intuitiv könnte man annehmen, dass dieses Problem so schwierig nicht sein könne, da man die Methode einfach ausführen könnte und warten könnte, ob sie irgendwann einen Wert zurückgibt. Bei genauem Hinschauen hat dieser Ansatz aber einen entscheidenden Haken: Woher wissen wir, ob die Methode zu einem späteren Zeitpunkt noch anhalten wird, oder ob das nie der Fall sein wird. Das heißt, dass das beschriebene Verfahren eine einseitige Entscheidung zulässt: Wenn die Methode irgendwann terminiert, können wir mit dem zuvor beschriebenen Verfahren eine positive Rückmeldung geben, im Negativfall werden wir aber keine Antwort erhalten. Ein solches Verfahren nennen wir ein Semientscheidungsverfahren und ein Problem, für das es ein Semientscheidungsverfahren gibt, semientscheidbar.

Den Einstieg in die Unentscheidbarkeit nehmen wir mit dem speziellen Halteproblem:

Definition 6.1.1 : Spezielles Halteproblem

Es sei Σ die Menge der Zeichen, aus denen ein Decision-C#-Code bestehen darf, dann enthält Σ^* alle Decision-C#-Quellcodes. Das spezielle Halteproblem K ist die Menge aller Entscheidungsmethoden, die, wenn man ihre eigene Codierung übergibt, termi-

nieren, formal:

$$K = \{\mathsf{w} \in \Sigma^* \mid \mathsf{w} \text{ ist eine Entscheidungsmethode und hält auf } \mathit{code}_\Sigma(\mathsf{w})\}$$

Wir werden mit einem Widerspruchsbeweis zeigen, dass das spezielle Halteproblem nicht entscheidbar ist.

Satz 6.1.2

Das spezielle Halteproblem K ist nicht entscheidbar, das heißt es gibt keine Decision-C#-Entscheidungsmethode, die K entscheidet.

Beweis. Wir zeigen diesen Satz mit einem Widerspruchsbeweis. Das heißt, wir nehmen zunächst an, dass K entscheidbar ist und führen diese Annahme zu einem logischen Widerspruch.

Sei also eine Decision-C#-Entscheidungsmethode k gegeben, die K entscheidet, das heißt $\mathsf{k}(\mathtt{w}) = \mathsf{true}$ genau dann, wenn $\mathit{decode}_\Sigma(\mathtt{w})$ eine Decision-C#-Entscheidungsmethode w ist, die auf der Eingabe `w` hält und $\mathsf{k}(\mathtt{w}) = \mathsf{false}$ anderenfalls.

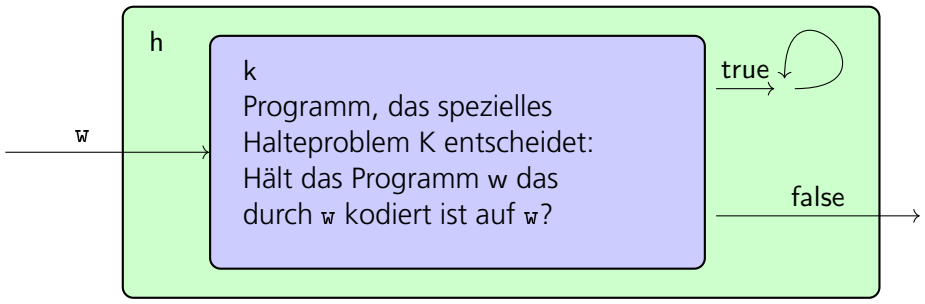
Wir konstruieren mittels der Entscheidungsmethode k eine neue Entscheidungsmethode h wie folgt:

```
bool h(int w)
{
    bool y = k(w);
    if(y)
    {
        while(true);
    }
    return false;
}
```

Wenn wir die Methode h für eine Eingabe `w` analysieren, stellen wir fest:

- Der Aufruf $\mathsf{h}(\mathtt{w})$ terminiert genau dann wenn der Rückgabewert von $\mathsf{k}(\mathtt{w})$ false ist.
- Wenn $\mathsf{h}(\mathtt{w})$ terminiert, dann bedeutet das nach Definition von k also, dass $\mathsf{w} = \mathit{decode}_\Sigma(\mathtt{w})$ (keine Entscheidungsmethode ist oder) bei der Eingabe `w` nicht terminiert.

Die Überlegung kann auch anhand der folgenden grafischen Darstellung von h nachvollzogen werden:



Da h eine Decision-C#-Entscheidungsmethode ist, können wir h über Σ codieren. Wir setzen $\mathtt{h} = \mathit{code}_\Sigma(\mathtt{h})$ und überlegen uns, ob der Aufruf $\mathtt{h}(\mathtt{h})$ terminiert.

- Wenn $\mathtt{h}(\mathtt{h})$ terminiert (und die Ausgabe false erzeugt), bedeutet das, dass $\mathtt{y} = \mathtt{false}$ gelten muss. Das wiederum bedeutet, dass $\mathit{decode}_\Sigma(\mathtt{h}) = \mathtt{h}$ (keine Entscheidungsmethode ist oder) bei der Eingabe h nicht terminiert. Da h nach Definition eine Entscheidungsmethode ist, folgt also: $\mathtt{h}(\mathtt{h})$ terminiert *nicht*. Das ist aber ein Widerspruch zur Annahme, dass $\mathtt{h}(\mathtt{h})$ terminiert.

- Wenn $\mathtt{h}(\mathtt{h})$ *nicht* terminiert, bedeutet das, dass $\mathtt{y} = \mathtt{true}$ gelten muss. Das wiederum bedeutet, dass $\mathit{decode}_\Sigma(\mathtt{h}) = \mathtt{h}$ bei der Eingabe h terminiert. Das ist aber ein Widerspruch zur Annahme, dass $\mathtt{h}(\mathtt{h})$ nicht terminiert.

Sowohl die Annahme, dass h terminiert, als auch die Annahme, dass h nicht terminiert, führen zu einem Widerspruch. Somit kann es die Methode h nicht geben. Das kann allerdings nur der Fall sein, wenn k nicht existiert, denn der übrige Code von h ist explizit angegeben. □

Der Beweis für die Unentscheidbarkeit des speziellen Halteproblems kann auch aufgefasst werden als ein Diagonalisierungsbeweis. Hierzu trägt man tabellarisch alle Decision-C#-Programme gegen ihre Codierungen ab und trägt in der Tabelle an dem Eintrag für Programm P und Codierung w ein, ob P auf w hält. Das aus der Diagonale konstruierte Programm h terminiert für jede Eingabe w genau dann wenn das durch w codierte Programm P_w nicht terminiert.

Wir haben also gesehen, dass nicht jedes Problem entscheidbar ist, man könnte aber durchaus mit einiger Berechtigung anmerken, dass das spezielle Halteproblem seinem Namen absolut gerecht wird und ein sehr spezielles Problem ist, an dessen Lösung man im Allgemeinen nicht oft interessiert ist. Daher werden wir uns in den folgenden beiden Abschnitten zunächst darum kümmern, eine Beweistechnik einzüben, mit der man für viele weitere Probleme unter Rückgriff auf bereits als unentscheidbar nachgewiesene Probleme zeigen kann, dass diese unentscheidbar sind. Schließlich werden wir mit dem Satz von Rice ein Ergebnis kennenlernen, das illustriert, wie weitverbreitet die Unentscheidbarkeit bei der Untersuchung von allgemeinen Decision-C#-Programmen – und damit auch allgemeinen C#-Programmen – ist.

Bemerkung 6.1.3

Zur Einordnung des Ergebnisses dieses Kapitels sei noch angemerkt, dass die Unentscheidbarkeit des speziellen Halteproblems an der Vereinfachung hängt, dass wir einen unendlichen Speicherplatz angenommen haben. Wenn man berücksichtigt, dass reale Computer nur endlich viel Speicherplatz zur Verfügung haben, könnte man streng genommen festhalten, dass das spezielle Halteproblem für jede feste Wahl von beschränkten C#-Programmen grundsätzlich entscheidbar wäre, weil nur endlich viele mögliche Speicherkonfigurationen möglich sind. Diese Einschränkung ist aber gleich in zweierlei Hinsicht akademischer Natur: Einerseits zeigt der Beweis dann immernoch, dass das allgemeine Entscheiderprogramm nicht auf diesem Computersystem selbst laufen könnte, andererseits ist bei jedem Computer von nennenswerter Speicherkapazität der Ansatz, sämtliche mögliche Konfigurationen erschöpfend zu untersuchen, nicht zielführend, weil der Aufwand so hoch wäre, dass das Anwender die Antwort nicht mehr in seiner Lebenszeit zu erwarten hätte.

6.1.2 Reduktion und weitere Halteprobleme

Bemerkung 6.1.4

Ab hier werden wir im Sinne der einfacheren Lesbarkeit nicht mehr in jedem Fall ausdrücklich zwischen einem Wort w über einem Alphabet Σ und seiner Codierung `w` unterscheiden. Das heißt, dass wir die Codierung und Decodierung implizit vornehmen und nicht mehr aufschreiben, wenn in einer Methode ein Wort zunächst als natürliche Zahl codiert oder eine natürliche Zahl als Wort decodiert werden muss. Als Lesehilfe nutzen wir für die Codierung das Zeichen `w` und für das Wort das Zeichen w, sowie bei Methoden und den durch sie implementierten Funktionen das Zeichen f für die Methode und das Zeichen f für die Funktion.

Um weitere Probleme als unentscheidbar nachzuweisen, wäre es wünschenswert, nicht wieder, wie im Fall des speziellen Halteproblems, einen elementaren Beweis zu führen, sondern uns den Umstand zu Nutze zu machen, dass wir bereits ein unentscheidbares Problem kennen. Die Idee hinter der Beweisführung, die wir beabsichtigen, ist die folgende:

1. Gegeben sei ein Problem P, von dem wir nachweisen wollen, dass es unentscheidbar ist.

2. Wir wollen wiederum einen Widerspruchsbeweis führen, um zu zeigen, dass P unentscheidbar ist.

3. Hierzu wählen wir ein (geeignetes) Problem U, von dem wir bereits wissen, dass es unentscheidbar ist.

4. Wir nehmen an, dass P entscheidbar wäre und führen diese Annahme zu einem Widerspruch.
5. Hierzu entwickeln wir ein Verfahren, wie wir das Problem U lösen können, indem wir die hypothetische Lösung von P aus Schritt 4 nutzen.
6. Gelingt es uns, ein Verfahren zu finden, das U mithilfe eines Entscheidungsverfahrens für P zu entscheiden vermag, dann folgt, dass es ein Entscheidungsverfahren für P nicht geben kann, weil wir bereits wissen, dass U nicht entscheidbar ist.

Der entscheidende Schritt für weitere Unentscheidbarkeitsbeweise sind also die Schritte 3 und 5 aus dem obigen Verfahren. Für Schritt 5 werden wir üblicherweise das Verfahren der Reduktion verwenden. Bei der Reduktion eines Problems U auf ein Problem P, in Zeichen $U \leq P$ führt man ein Entscheidungsverfahren für U auf ein Entscheidungsverfahren für P zurück.

Definition 6.1.5 : Reduktion

Gegeben seien zwei Entscheidungsprobleme $A \subseteq \Sigma_1^*$ und $B \subseteq \Sigma_2^*$. Eine berechenbare Funktion

$$f\colon \Sigma_1^* \to \Sigma_2^*$$

nennen wir eine Reduktionsfunktion von A auf B, falls für alle $w \in \Sigma_1^*$ gilt:

$$w \in A \Leftrightarrow f(w) \in B.$$

Wir schreiben dann $A \leq B$.

Die Schreibweise $A \leq B$ in dieser Definition rührt daher, dass A im Fall dessen, dass es sich auf B reduzieren lässt, leichter als (oder gleich schwer wie) B hinsichtlich Entscheidbarkeit ist, das heißt, dass aus der Entscheidbarkeit von B folgt, dass auch A entscheidbar ist und aus der Unentscheidbarkeit von A folgt, dass auch B unentscheidbar ist. Wir halten das in einem Lemma fest:

Lemma 6.1.6

Es sei $A \subseteq \Sigma_1^*$, $B \subseteq \Sigma_2^*$ und $A \leq B$ vermöge der Reduktionsfunktion $f\colon \Sigma_1^* \to \Sigma_2^*$. Dann gilt:

1. Falls B entscheidbar ist, folgt, dass auch A entscheidbar ist.
2. Falls A unentscheidbar ist, dann ist auch B unentscheidbar.

Beweis. Wir zeigen die beiden Aussagen separat:

1. Sei B mittels der Entscheidungsmethode b entscheidbar, dann konstruieren wir eine Entscheidungsmethode a für A wie folgt:

```
bool a(int w)
{
    int y = f(w);
    if(b(y))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Die Methode a gibt true zurück genau dann wenn b(f(w)) den Wert true zurückgibt. Nach Definition von f gilt:

$$\mathsf{a}(\mathtt{w}) = \mathsf{true} \Leftrightarrow \mathsf{b}(\mathsf{f}(\mathtt{w})) = \mathsf{true} \Leftrightarrow f(w) \in B \Leftrightarrow w \in A$$

Also können wir mit a das Problem A entscheiden.

2. Sei A unentscheidbar, dann ist auch B unentscheidbar, das zeigen wir mit einem Widerspruchsbeweis. Wir nehmen also an, B sei entscheidbar mittels einer Entscheidungsmethode b, dann können wir A mit der folgenden Entscheidungsmethode a entscheiden:

```
bool a(int w)
{
    int y = f(w);
    if(b(y))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Die Methode a gibt true zurück genau dann wenn b(f(w)) den Wert true zurückgibt. Nach Definition von f gilt:

$$\mathsf{a}(\mathtt{w}) = \mathsf{true} \Leftrightarrow \mathsf{b}(\mathsf{f}(\mathtt{w})) = \mathsf{true} \Leftrightarrow f(w) \in B \Leftrightarrow w \in A$$

Also könnten wir mit a das Problem A entscheiden, was im Widerspruch zur Annahme steht, dass A unentscheidbar ist. Also kann b nicht existieren und B ist unentscheidbar.

□

Die beiden Beweisteile aus obigem Lemma sind nahezu vollständig analog, nur die Argumentation am Ende ändert sich abhängig davon, ob wir von einem entscheidbaren Problem B oder einem unentscheidbaren Problem A ausgehen. Für unsere Zwecke in diesem Kapitel benötigen wir die zweite Aussage des Lemmas, wohingegen die erste Aussage keine Verwendung finden wird. Allerdings kann man die erste Aussage grundsätzlich als Problemlösungsstrategie in der Software-Entwicklung nutzen. Wenn man bereits ein funktionierendes Verfahren für ein Problem B hat und eine Reduktionsfunktion für ein neues Problem A auf B findet, kann man mit verhältnismäßig geringem Aufwand eine Lösung für Problem A umsetzen.

Das Beweisverfahren der Reduktion wollen wir jetzt für das allgemeine Halteproblem *H* anwenden. Dieses Problem ist sehr nah an dem speziellen Halteproblem, mit dem einzigen Unterschied, dass uns das Verhalten auf einer beliebigen Eingabe, statt nur auf der eigenen Codierung interessiert.

Definition 6.1.7 : Allgemeines Halteproblem

Es sei Σ die Menge der Zeichen, aus denen ein Decision-C#-Code bestehen darf. Das allgemeine Halteproblem ist die Menge aller Paare von Entscheidungsmethoden m und Eingaben w, so dass m, wenn man ihr w übergibt, terminiert. Formal:

$$H = \{(\mathsf{m}, \mathsf{w}) \in \Sigma^* \times \mathbb{Z} \mid \mathsf{m} \text{ ist eine Entscheidungsmethode und hält auf } \mathsf{w}\}$$

Mithilfe von Reduktion können wir zeigen, dass das allgemeine Halteproblem unentscheidbar ist:

Satz 6.1.8

Das allgemeine Halteproblem H ist unentscheidbar.

Beweis. Wir zeigen dies mit einer Reduktion des speziellen Halteproblems K auf H, das heißt, wir zeigen, dass wenn wir ein Entscheidungsverfahren h für H finden könnten, wir auch ein Entscheidungsverfahren k für K konstruieren könnten. In Zeichen ist zu zeigen $K \leq H$.

Zu diesem Zweck nehmen wir an, dass es eine Entscheidungmethode h gibt, die H entscheidet und zwei Parameter erwartet, die Zahl m die den Quellcode der zu untersuchenden Methode m codiert und die Eingabe w, auf der m untersucht werden soll.

```
bool k(int w)
{
    bool y = h(w, w);
    return y;
}
```

Offensichtlich ist k eine zulässige Methode, sofern die Methode h existiert. Die Reduktionsfunktion $f\colon \Sigma^* \to \Sigma^* \times \mathbb{Z}$ hat also die Form $f(w) = (w, \mathit{code}_\Sigma(w))$. Zu zeigen ist nun, dass f eine korrekte Reduktionsfunktion ist:

$$\begin{aligned} & w \in K \Leftrightarrow w \text{ ist eine Entscheidungsmethode und hält auf } \mathit{code}_\Sigma(w) \\ \Leftrightarrow & (w, \mathit{code}_\Sigma(w)) \in H \Leftrightarrow f(w) \in H \end{aligned}$$

Also haben wir gezeigt, $K \leq H$ und zusammen mit dem vorherigen Ergebnis, dass K unentscheidbar ist, folgt, dass H unentscheidbar ist. □

Eine weitere bekannte Variante des Halteproblems ist das Halteproblem bei leerer Eingabe – und ganz analog bei jeder beliebigen festen Eingabe.

Definition 6.1.9 : Halteproblem bei leerer Eingabe

Es sei Σ die Menge der Zeichen, aus denen ein Decision-C#-Code bestehen darf. Das Halteproblem bei leerer Eingabe ist die Menge aller Entscheidungsmethoden m, so dass m, wenn man ihr 0, also die Codierung des leeren Strings, übergibt, terminiert. Formal:

$$H_0 = \{\mathsf{m} \in \Sigma^* \mid \mathsf{m} \text{ ist eine Entscheidungsmethode und hält auf } 0\}$$

Bevor wir die Unentscheidbarkeit von H_0 beweisen, soll die folgende Aufgabe dazu dienen, eine sehr häufige Fehlvorstellung zu verhindern und für einen wichtigen Stolperstein bei Reduktionsbeweisen zu sensibilisieren:

Selbsttestaufgabe 6.1.1

Nach dem vorhergehenden Beweis für das allgemeine Halteproblem und der Definition des Halteproblems bei leerer Eingabe mag einem Leser, das keine vorherigen Kenntnisse in der Theoretischen Informatik hat, eine einfache Idee für den Beweis der Unentscheidbarkeit von H_0 kommen: Man übergebe zur Konstruktion einer Methode h_0, die das H_0 entscheidet, wie im Beweis für das allgemeine Halteproblem an die Methode h die eingegebene Methode und als zweiten Parameter die 0. Mit dieser Reduktionsfunktion f gilt: $f(w) \in H \Leftrightarrow w \in H_0$. Wieso ist dieser Beweis nicht dazu geeignet, um nachzuweisen, dass H_0 unentscheidbar ist?

Musterlösung zu Selbsttestaufgabe 6.1.1

Bei der Beweisskizze in der Aufgabenstellung wurde die Reduktionsrichtung vertauscht. Gezeigt wurde, dass $H_0 \leq H$ gilt. Wir wissen aber bislang nur von H, nicht von H_0, dass es unentscheidbar ist. Um zu beweisen, dass H_0 unentscheidbar ist, müssten wir zeigen $H \leq H_0$. Nachdem wir bereits wissen, dass H unentscheidbar ist,

ist ein Nachweis, dass $H_0 \leq H$ richtig ist, nicht instrumentell.

Da wir uns noch einmal vor Augen geführt haben, wie ein Beweis der Unentscheidbarkeit für H_0 strukturiert sein muss, können wir den Beweis jetzt ausführen.

Satz 6.1.10

Das Problem H_0 ist unentscheidbar.

Beweis. Wir wollen zeigen, dass $H \leq H_0$ gilt, dass wir also das allgemeine Halteproblem mittels des Halteproblems bei leerer Eingabe entscheiden können. Sei hierzu h0 eine Entscheidungsmethode, die H_0 entscheidet. Weiterhin sei f eine Berechnungsmethode die alle Zahlen, die keine Entscheidungsmethode codieren, einfach durchreicht und sonst wie folgt funktioniert:

- Gegeben sind eine Entscheidungsmethode m und eine Eingabe x; wir wollen wissen, ob m auf x terminiert.
- $f(\mathsf{m}, \mathtt{x}) = \mathsf{nx}$, wobei nx die folgende Entscheidungsmethode ist:

```
bool nx(int w)
{
    return m(x);
}
```

Man beachte, dass nx ihren Parameter w einfach ignoriert und sich immer wie m auf x verhält – also insbesondere auch für $\mathtt{w} = 0$. Die Implementation f von f hängt im Wesentlichen an Codierung und Decodierung von m und würde keinen Erkenntnisgewinn ergeben, so dass wir an der Stelle auf die explizite Angabe des Codes von f verzichten.

Für f gilt dann:

$$(\mathsf{m}, \mathtt{x}) \in H \Leftrightarrow \mathsf{m}(\mathtt{x}) = \mathsf{true} \Leftrightarrow \mathsf{nx}(0) = \mathsf{true} \Leftrightarrow \mathsf{h0}(f(\mathsf{m}, \mathtt{x})) = \mathsf{true} \Leftrightarrow f(\mathsf{m}, \mathtt{x}) \in H_0$$

Wir könnten also, wenn wir H_0 entscheiden könnten, auch H entscheiden. Da H aber unentscheidbar ist, muss auch H_0 unentscheidbar sein. □

Zum Abschluss dieses Kapitels betrachten wir noch eine Variation des Halteproblems, dessen Unentscheidbarkeit auf eine ähnliche Weise nachgewiesen werden kann wie die von H_0:

Selbsttestaufgabe 6.1.2

Wir betrachten das Problem H_{0s}, das strikte Halteproblem bei Eingabe 0, das wir wie folgt definieren:

$$H_{0s} = \{\mathsf{m} \in \Sigma^* \mid \mathsf{m} \text{ ist eine Entscheidungsmethode und hält ausschließlich auf } 0\}$$

1. Zeigen Sie, dass $H_0 \leq H_{0s}$ ist.
2. Zeigen Sie, dass auch $H_0 \leq \overline{H_{0s}}$ gilt.

Musterlösung zu Selbsttestaufgabe 6.1.2

1. Wir zeigen $H_0 \leq H_{0s}$. Das sieht man ein mit der Reduktionsfunktion f die alle Zahlen, die keine Entscheidungsmethode codieren, einfach durchreicht und sonst wie folgt funktioniert:
   - Gegeben sei eine Entscheidungsmethode m; wir wollen wissen, ob m auf 0 terminiert.
   - $f(\mathsf{m}) = \mathsf{n}$, wobei n die folgende Methode ist:

```
bool n(int x)
{
    while(x != 0);
    return m(x);
}
```

   Das Verhalten von n ist folgendes: Falls die Eingabe x nicht 0 ist, verbleibt n in einer Endlosschleife, anderenfalls wird m(0) zurückgegeben. Die Methode n terminiert also auf keiner Eingabe außer 0 und terminiert bei Eingabe 0 dann und nur dann wenn m auf der Eingabe 0 terminiert.

2. Wir beginnen damit, uns zu überlegen, welche Menge $\overline{H_{0s}}$ ist: Zunächst enthält $\overline{H_{0s}}$ alle Zeichenketten, die nicht die Kodierung einer Entscheidungsmethode sind. Zusätzlich enthält die Menge alle Zeichenketten, die die Kodierung einer Entscheidungsmethode m sind, die jedenfalls eine der folgenden zwei Bedingungen erfüllen:
   - m hält nicht auf der Eingabe 0.
   - m hält für eine Eingabe, die nicht 0 ist.

   Mit dieser Vorabüberlegung zeigen wir nun wieder $H_0 \leq \overline{H_{0s}}$. Hierzu verwenden wir die folgende Reduktionsfunktion f, die alle Zahlen, die keine Entscheidungsmethode codieren, auf eine Methode g abbildet, die exakt bei der Eingabe 0 terminiert:

```
bool g(int x)
{
    while(x != 0);
    return true;
}
```

   und sonst wie folgt funktioniert:

- Gegeben sei eine Entscheidungsmethode m; wir wollen wissen, ob m auf 0 terminiert.
- $f(\mathsf{m}) = \mathsf{n}$, wobei n die folgende Methode ist:

```
bool n(int x)
{
    while(x == 0);
    return m(0);
}
```

Die Methode n hat nun die folgenden Eigenschaften:

- Falls sie die Eingabe 0 übergeben bekommt, verbleibt sie in einer Endlosschleife und terminiert nicht.
- Bekommt sie irgendeine andere Eingabe, gibt sie $\mathsf{m}(0)$ zurück; sie terminiert für jede Eingabe außer 0 also dann und nur dann, wenn m auf 0 terminiert.

Insgesamt gilt also: $\mathsf{n} = \mathsf{f}(\mathsf{m})$ ist in H_{0s} genau dann, wenn m in H_0 liegt.

Diese Selbsttestaufgabe demonstriert also auch, dass es unentscheidbare Probleme gibt, die nicht einmal semientscheidbar sind.

6.1.3 Der Satz von Rice und weitere unentscheidbare Probleme

Bislang haben wir zwar bereits eine Reihe von unentscheidbaren Problemen kennengelernt, die waren allerdings alle konzeptuell sehr ähnlich, es ging nämlich stets darum, ob eine gegebene Decision-C#-Entscheidungsmethode terminiert oder nicht. Der Satz von Rice macht allerdings deutlich, dass unentscheidbare Probleme nicht etwa ein akademischer Sonderfall sind, sondern, sobald man C#-Programme analysieren möchte, ein omnipräsentes Hindernis darstellen.

Satz 6.1.11 : Satz von Rice

Es sei $\mathcal{S}$ die Menge aller durch Decision-C#-Berechnungsmethoden berechenbaren (partiellen) Funktionen und $\mathcal{R} \subseteq \mathcal{S}$ eine nicht triviale Teilmenge von $\mathcal{S}$, das heißt:

- $\mathcal{R} \neq \emptyset$, d.h. $\mathcal{R}$ enthält mindestens eine Funktion.
- $\mathcal{R} \neq \mathcal{S}$, d.h. $\mathcal{R}$ enthält mindestens eine Funktion aus $\mathcal{S}$ *nicht*.

Dann ist unentscheidbar, ob eine gegebene Decision-C#-Berechnungsmethode eine Funktion berechnet, die in $\mathcal{R}$ liegt.

Bemerkung 6.1.12

Der triviale Fall ist im Satz von Rice in der Tat entscheidbar. Ist $\mathcal{R} = \mathcal{S}$ muss eine Entscheidungsmethode immer `true` ausgeben, ist $\mathcal{R} = \emptyset$, muss sie immer `false` ausgeben, um zu entscheiden, ob eine gegebene Decision-C#-Berechnungsmethode eine Funktion aus $\mathcal{R}$ berechnet.

Für den Beweis nutzen wir noch ein einfaches Lemma:

Lemma 6.1.13

Es sei $P \subseteq \Sigma^*$ unentscheidbar, dann ist auch $\overline{P} = \Sigma^* \setminus P$ unentscheidbar.

Beweis. Es sei P unentscheidbar. Nehmen wir nun an $\overline{P}$ wäre entscheidbar, es gäbe also eine Entscheidungsmethode `q`, die $\overline{P}$ entscheidet. Dann entscheidet die folgende Methode `p` das Problem P:

```
int p(int x)
{
    if(q(x)) return false;
    else return true;
}
```

□

Damit können wir nun den Satz von Rice beweisen:

Beweis. (zum Satz von Rice) Gegeben sei eine nicht-triviale Menge $\mathcal{R} \subset \mathcal{S}$ und wir definieren das folgende Entscheidungsproblem:

$$R = \{\mathsf{m} \in \Sigma^* \mid \mathsf{m} \text{ ist eine Berechnungsmethode und berechnet eine Funktion in } \mathcal{R}\}$$

Wir führen eine Fallunterscheidung danach durch, ob die überall undefinierte Funktion Ω in $\mathcal{R}$ liegt oder nicht.

- Falls $\Omega \in \mathcal{R}$, so muss es mindestens eine (partielle) Funktion $g \in \mathcal{S} \setminus \mathcal{R}$ geben, weil $\mathcal{R}$ nicht trivial ist. Nach Definition von $\mathcal{S}$ bedeutet das, dass es eine Decision-C#-Berechnungsmethode `g` geben muss, die g berechnet.

  Mit dieser Vorüberlegung können wir nun H_0 auf $\overline{R}$ reduzieren. Hierzu betrachten wir die folgende Reduktionsfunktion f:

  - Gegeben sei eine Entscheidungsmethode `m`; wir wollen wissen, ob `m` auf 0 terminiert.
  - $f(\mathsf{m}) = \mathsf{n}$, wobei `n` die folgende Berechnungsmethode ist:

```
int n(int x)
{
    m(0);
    return g(x);
}
```

Welche Funktion berechnet n nun? Falls m auf 0 anhält, berechnet n gerade g. Anderenfalls ist m die überall undefinierte Funktion Ω. Es gilt also:

$$\mathsf{m} \in H_0 \Leftrightarrow f(\mathsf{m}) = \mathsf{g} \Leftrightarrow f(\mathsf{m}) \notin R \Leftrightarrow f(\mathsf{m}) \in \overline{R}$$

Wir sehen also, dass $\overline{R}$ nicht entscheidbar ist und somit (mit Lemma 6.1.13) ist auch R nicht entscheidbar.

- Ganz analog kann man verfahren für den Fall dass $\Omega \notin \mathcal{R}$, also $\Omega \in \mathcal{S} \setminus \mathcal{R}$: Wenn $\Omega \in \mathcal{S} \setminus \mathcal{R}$, dann muss es eine (partielle) Funktion $g \in \mathcal{R}$ geben (die offensichtlich nicht überall undefiniert ist), weil $\mathcal{R}$ nicht trivial gewählt war. Es muss eine Decision-C#-Berechnungsmethode g geben, die g berechnet. Damit können wir H_0 auf R reduzieren mit genau der gleichen Funktion f wie im ersten Schritt und beobachten, dass n die Funktion g berechnet, wenn m auf 0 anhält und Ω sonst. Es gilt also:

  $$\mathsf{m} \in H_0 \Leftrightarrow f(\mathsf{m}) = \mathsf{g} \Leftrightarrow f(\mathsf{m}) \in R$$

  Wir haben also gezeigt $H_0 \leq R$ und damit ist R nicht entscheidbar.

□

Der Satz von Rice ist ein sehr mächtiges Instrument, um mit wenig Aufwand die Unentscheidbarkeit von vielen Eigenschaften zu zeigen und zeigt ein fundamentales Problem für die automatische Verifikation auf, deren Ziel es gerade ist, festzustellen, dass ein Programm oder eine Methode ein bestimmtes Verhalten zeigt.

Beispiel 6.1.14

Wir betrachten die konstante 1-Funktion $f\colon \mathbb{Z} \to \mathbb{Z}, f(x) = 1$ und zeigen mithilfe des Satzes von Rice, dass nicht entscheidbar ist, ob eine Berechnungsmethode die Funktion f berechnet. Hierzu setzen wir:

$$\mathcal{R} = \{f\} \subset \mathcal{S}$$

Wir sehen, dass es mindestens eine Funktion in $\mathcal{R}$ gibt, nämlich gerade f, und mindestens eine Funktion in $\mathcal{S} \setminus \mathcal{R}$, beispielsweise die überall undefinierte Funktion Ω. Es ist ersichtlich, dass f auch tatsächlich berechenbar ist, denn die folgende Berechnungsmethode berechnet offensichtlich f:

```
int f(int x)
{
    return 1;
```

```
}
```

Also ist $\mathcal{R}$ eine nicht triviale Teilmenge von $\mathcal{S}$ und nach dem Satz von Rice ist nicht entscheidbar, ob eine Berechnungsmethode die konstante 1-Funktion berechnet.

Selbsttestaufgabe 6.1.3

Welche der folgenden Fragestellungen sind unentscheidbar? Beweisen Sie entweder mit dem Satz von Rice, dass die Fragestellung nicht entscheidbar ist, oder begründen Sie, wieso der Satz von Rice nicht anwendbar ist.

1. Berechnet eine Berechnungsmethode eine totale Funktion?
2. Gibt eine Berechnungsmethode immer, wenn sie terminiert, eine Primzahl zurück?
3. Enthält die Berechnungsmethode eine Multiplikation?

Musterlösung zu Selbsttestaufgabe 6.1.3

1. Wir beobachten, dass es totale Funktionen gibt, die berechenbar sind (Beispiel: Die konstante 1-Funktion) und berechenbare Funktionen gibt, die nicht total sind (Beispiel: Ω) Also ist

   $$\mathcal{R} = \{f : \mathbb{Z} \to \mathbb{Z} \mid f \text{ ist total}\}$$

   eine echte, nicht-leere Teilmenge von $\mathcal{S}$ und es ist somit nicht entscheidbar, ob eine Berechnungsmethode eine Funktion aus $\mathcal{R}$ berechnet.

2. Wir beobachten, dass es Funktionen gibt, die immer wenn sie terminieren eine Primzahl zurückgeben und berechenbar sind (Beispiel: Ω) und berechenbare Funktionen gibt, die nicht immer wenn sie terminieren eine Primzahl zurückgeben (Beispiel: Die konstante 1-Funktion) Also ist

   $$\mathcal{R} = \{f : \mathbb{Z} \to \mathbb{Z} \mid \forall x \in \mathbb{Z} : f(x) \text{ ist definiert } \Rightarrow f(x) \text{ ist prim}\}$$

   eine echte, nicht-leere Teilmenge von $\mathcal{S}$ und es ist somit nicht entscheidbar, ob eine Berechnungsmethode eine Funktion aus $\mathcal{R}$ berechnet.

3. Die hier angegebene Eigenschaft ist eine Eigenschaft der Methode selbst und schränkt die berechnete Funktion einer Berechnungsmethode nicht ein. In der Tat gibt es beispielsweise zwei verschiedene Decision-C#-Berechnungsmethoden, die beide Ω berechnen; die eine enthält eine Multiplikation, die andere nicht:

   ```
   int omega1(int x)
   ```

```
{
    while(true);
    return 0;
}
int omega2(int x)
{
    int y = 2 * x;
    while(true);
    return 0;
}
```

Die Eigenschaft ist darüber hinaus durch einfaches Überprüfen des Quelltextes entscheidbar.

Zum Abschluss unseres Überblicks zum Thema Berechenbarkeitstheorie geben wir noch einen kurzen Überblick über weitere bekannte unentscheidbare Probleme, für die wir hier aber keine Beweise angeben werden.

Definition 6.1.15 : Postsches Korrespondenzproblem

Gegeben sei eine endliche Menge von Wortpaaren $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ über einem gemeinsamen Alphabet Σ. Zu entscheiden ist, ob es eine Indexfolge $i_1, i_2, ..., i_m$ gibt, so dass $x_{i_1} x_{i_2} ... x_{i_m} = y_{i_1} y_{i_2} ... y_{i_m}$ gibt.

Zusätzlich sind viele Fragestellungen zu kontextfreien Grammatiken unentscheidbar. Was eine kontextfreie Grammatik ist, werden wir in der nächsten Lektion besprechen, an der Stelle sei schon einmal vorgegriffen, dass es unentscheidbar ist, ob der Schnitt der Sprachen zweier kontextfreier Grammatiken ebenfalls kontextfrei ist oder ob dieser Schnitt leer ist.

6.2 Komplexitätsklassen

In unseren bisherigen Überlegungen haben wir uns darauf konzentriert, ob eine Fragestellung grundsätzlich automatisiert entschieden werden kann, doch in aller Regel ist die reine Entscheidbarkeit, wenn man an der Lösung eines Problems interessiert ist, nicht ausreichend. Es bringt einem Nutzer nichts, wenn es weiß, dass es die Antwort auf seine Fragestellung grundsätzlich irgendwann erhalten würde, die erwartete Zeit bis zum Erhalt der Antwort aber die Lebenszeit des Nutzers übersteigt. In diesem Abschnitt befassen wir uns mit Komplexitätsklassen, die es uns ermöglichen, einzuordnen, wie schwierig ein Problem ist und die es uns erlauben, Probleme grob in effizient entscheidbare und (wahrscheinlich) nicht effizient entscheidbare Probleme einzuteilen.